LaBRI

université de BORDEAUX

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE
MATHEMATIQUES ET INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

par **Gabriel Bathie**

Algorithmes d'approximation efficaces pour les langages formels

---

Fast Approximation Algorithms for Formal Languages

Sous la direction de :

Nathanaël Fijalkow
et **Tatiana Starikovskaya**

Soutenue le 12 Juin 2025
Devant la comission d'examen composée de :

M. Michal Koucký	Professor, Charles University	Rapporteur
M. Frédéric Magniez	Directeur de Recherche, IRIF	Rapporteur
Mme Inge Li Gørtz	Professor, DTU	Examinatrice
M. Charles Paperman	Maître de Conférences, Université de Lille	Examinateur
Mme Raluca Uricaru	Maîtresse de Conférences, Université de Bordeaux	Examinatrice
M. Nathanaël Fijalkow	Directeur de Recherche, LaBRI	Directeur

Membre invitée:

Mme Tatiana Starikovskaya	Maîtresse de Conférences, ENS Ulm	Co-encadrante

Abstract

Since the 70's, computer scientists have developed very efficient tools and algorithms for text processing. However, these techniques are mostly focused on *exact* problems, and cannot be straightforwardly adapted to *approximation* tasks, or do not scale to the size of modern datasets. In this thesis, we study resource-efficient approximation algorithms for text processing.

The first part of this thesis studies variants of approximate pattern matching, tasks where one must find all substrings of a text that are *similar* to the given pattern, for various notions of similarity. We first investigate the case of *circular pattern matching*. We give a data structure with a time-space trade-off in the read-only model for *internal pattern matching*, and use it to solve circular pattern matching and the longest common substring problem with little additional space in the read-only model. Next, we consider the case of similarity measured by the Hamming distance, in strings with wildcards, that is, characters that match any other character. Focusing on the low-distance regime and on the case of strings with few contiguous groups of wildcards, we give an efficient algorithm and a combinatorial characterization of the structure of occurrences in this setting. We give a lower bound that shows that our characterization is close to optimal. Finally, we study data structures for computing longest common extensions in string with wildcards (LCEW). We give a data structure that provide a time-space trade-off in the read-only model, and use it to derive efficient algorithms for pattern matching and analysis of string with wildcards. We also show a connection to sparse Boolean matrix multiplication, from which we derive lower bounds for combinatorial data structures for LCEW.

The second part of this thesis studies the task of deciding approximate membership in a formal language, in three different frameworks. We start with the Property Testing, the framework of *information-efficient* algorithms, in our case for regular languages, following the seminal work of Alon, Krivelevich, Newman and Szegedy [SIAM J. Comp. 99]. We give a complete characterization of the complexity of testing regular languages under the Hamming distance, showing that any regular language belongs to one of three complexity classes, and that these classes can be characterized using combinatorial objects, called *minimal blocking sequences*. We further show that this characterization is effective: given a finite automaton, deciding to which class it belongs to is complete for PSPACE. Next, we turn to the question of computing the Hamming or edit distance between an input word and a given language. We give small-space algorithms in the read-only or streaming model for the low-distance version of these problems, for languages of palindromes and of squares. Finally, we design a small-space algorithm for computing the *palindromic length* of a string in the read-only model. Our algorithm is based on novel results on the structure of the *k-palindromic prefixes* of a string, i.e., its prefixes of palindromic length k.

Keywords: formal languages, pattern matching, streaming algorithms, property testing

Résumé

Depuis les années 1970, la recherche en informatique a produit de nombreux outils et algorithmes très efficaces pour le traitement de données textuelles. Toutefois, ces techniques ont principalement été conçues pour les problèmes *exacts*, et ne sont pas directement applicables aux tâches *d'approximation* ou ne sont pas adaptées à la taille des jeux de données modernes. Dans cette thèse, nous étudions des algorithmes d'approximation efficaces (en termes de ressources), qui sont plus adaptés aux tâches modernes de traitement de texte.

Dans la première partie de cette thèse, on s'intéresse à plusieurs variantes du problème de recherche de motif, un ensemble de tâche dans lesquelles on doit trouver toutes les sous-chaînes d'un texte qui sont *similaires* à un motif donné, pour plusieurs notions de similarité. On étudie tout d'abord le cas de la recherche *circulaire* de motif. Nous donnons une structure de données avec un compromis temps-espace mémoire pour le problème de recherche interne de motif, dans le modèle read-only, puis nous montrons comment l'utiliser pour obtenir des algorithmes pour le problème de recherche circulaire de motif et celui de calculer la plus longue sous-chaîne commune en utilisant un espace mémoire faible. Ensuite, on s'intéresse au cas où la similarité est définie par la distance de Hamming, et où les chaînes de caractères peuvent contenir des *jokers*, c'est-à-dire des caractères considérés égaux à n'importe quel autre caractère. Dans le cas particulier des distances faibles et où les chaînes contiennent un petit nombre de groupes contigus de jokers, nous donnons un algorithme efficace et une caractérisation combinatoire de la structure des occurrences dans ce contexte. Nous donnons également une borne inférieure qui montre que notre caractérisation est presque optimale. Enfin, on étudie les structure de données pour calculer les plus longues extensions communes dans les chaînes avec jokers (abrégé LCEW en anglais). Nous donnons une structure de données avec un compromis espace-temps dans le modèle read-only, puis nous montrons comment l'utiliser pour concevoir des algorithmes efficace pour la recherche de motif avec quelques erreurs et pour l'analyse des propriétés des chaînes contenant des jokers. Nous exhibons également une relation avec le problème de multiplication de matrices booléennes parcimonieuses, que nous utilisons pour donner des bornes inférieures sur la complexité de structures de données pour LCEW.

La seconde partie de cette thèse s'intéresse à la question d'approximer l'appartenance d'un mot à un langage formel, dans trois contextes différents. On commence par le modèle du property testing, modèle des algorithmes frugaux en *information*, dans notre cas pour les langages formels. Notre étude s'inscrit dans la lignée du travail fondateur d'Alon, Krivelevich, Newman et Szegedy [SIAM J. Comp. 99] sur le property testing des langages réguliers. Nous donnons une caractérisation complète de la complexité du property testing des langages réguliers pour la distance de Hamming, en montrant que tout langage régulier appartient à une de trois classes de complexité. Nous montrons également que le problème de classification associé est complet pour PSPACE. Ensuite, on étudie la question de calculer la distance de Hamming ou d'édition entre un mot et un langage formel donné. Nous donnons des algorithmes utilisant peu d'espace mémoire dans les modèles read-only et streaming, pour les version "faible distance" de ces questions, pour les langages des palindromes et des carrés. Enfin, nous donnons un algorithme efficace en espace permettant de calculer la longueur palindromique d'une chaîne dans le modèle read-only. Notre algorithme est basé sur des résultats nouveaux sur la structure des préfixes *k-palindromiques* d'un mot, c'est-à-dire ses préfixes de longueur palindromique k.

Mots-clés : langages formels, recherche de motif, algorithmes de streaming, property testing

Thèse préparée au **Laboratoire Bordelais de Recherche en Informatique (LaBRI)**
351, cours de la Libération, F-33405 Talence cedex

Acknowledgements

First, I would like to express my gratitude to Frédéric Magniez and Michal Koucký for accepting to review my manuscript and to be part of the jury. I would also like to thank Inge Li Gørtz, Charles Paperman and Raluca Uricaru for accepting to be part of the jury.

Les trois années de ma thèse ont été marquées par toutes les personnes que j'ai rencontrées, avec qui j'ai pu travailler, réfléchir, discuter, rire et passer de bons moments. J'aimerais les remercier maintenant.

En première ligne viennent mes encadrants de thèse, Tatiana et Nathanaël. Qu'ils soient ici remerciés pour leurs conseils, les connaissances qu'ils m'ont partagées et les échanges que nous avons eus au cours de (et avant) cette thèse. Tatiana, merci de m'avoir toujours poussé à donner le meilleur de moi-même, et ce depuis mon stage de master ! Nathanaël, merci de m'avoir encouragé à suivre ma curiosité et soutenu dans mes projets, aussi variés soient-ils !

I would also like to thank everyone I had the opportunity to work with during my thesis: Antoine Amarilli, Itai Boneh, Panagiotis Charalampopoulos, Jonas Ellert, Nathanaël Fijalkow, Roman Kniazev, Tomasz Kociumaka, Guillaume Lagarde, Corto Mascle, Théo Matricon, Mikaël Monet Baptiste Mouillon, Tatiana Starikovskaya, Pierre Vandenhove and Ryan Williams – thank you for everything I learned from you, there is a part of you in this thesis! Panos, thank you for hosting me in London: that was probably the most productive time of my thesis, and we still found time for pizza and Greek food. Je remercie également Antoine, Mikaël et les membres de l'équipe LINKS qui m'ont accueilli pour un court séjour à l'INRIA Lille. J'en garde un très bon souvenir, tant pour nos discussions qui nous ont emmenés jusqu'à des questions surprenantes sur les codes de Gray que pour les agréables soirées passées dans le centre de Lille.

Je tiens à remercier les habitants des "Hauts du DI" de l'ENS de la rue d'Ulm, où a commencé cette thèse, pour leur accueil, les jeux et les discussions lors des pauses café. Je veux aussi remercier ici les membres du LaBRI, où j'ai fini ma thèse. En particulier, je remercie les doctorants et membres de l'AFoDIB pour la vie de laboratoire exceptionnelle qu'ils insufflent au LaBRI, et les membres de M2F pour leur accueil au sein du département. Je dis également merci aux membres de équipes administratives de l'ENS et du LaBRI qui font tourner la machine de la recherche depuis les coulisses pour leur aide tout au long de cette thèse.

Je veux aussi remercier tous les amis que j'ai côtoyé pendant ma thèse. Les moments passés avec vous me sont précieux, plus peut-être que je ne le laisse paraître. Merci aux occupants passés et présents du Bureau 325, Théo, Elsa, Clara, Rémi (et Timothée Ch.), Françoise et Sarah, pour cet environnement, souvent plus propice aux ragots qu'au travail, mais toujours plein de bonne humeur, auquel vous avez participé. Des remerciements particuliers vont à Garance, ma "grande sœur de thèse", pour les échanges que nous avons eus et les conseils précieux qu'elle a donnés au jeune thésard un peu déboussolé que j'ai été. Merci Jérôme pour ta bonne humeur et nos discussions. Merci Pierre, Guillaume

et Guru pour votre compagnie pendant le voyage à Philadelphie. Merci Yanis pour tes invitations régulières à des pauses café et les discussions qui en ont découlé. Merci Rémi pour nos échanges sur la vie, le monde académique, et le reste (au fait, tu as déjà pensé à réécrire Knowledge en Rust et ta thèse en Typst ?). Merci Thibault pour les cookies et de t'assurer (entre deux mails) que les doctorants pensent à manger chaque midi. Merci Guillaume de m'écouter avec enthousiasme même quand je te demande "Mais qu'est-ce qui se passe si la hiérarchie polynomiale s'effondre très vite ?". Merci Zoé pour nos échanges et nos déjeuners entre asociaux. Merci Corto d'être une source infinie de problème ouverts et d'être responsable de la fermeture de quasiment autant de problèmes. Merci Diane pour les encouragements dans la course au permis, que tu as brillamment remportée. Merci Clément, Sarah, Théo et Zoé pour nos sorties escalade. Merci Théo de toujours être à l'écoute de mes idées, d'en acheter certaines et de démonter les autres. (Tu as adhéré au culte du meilleur langage de programmation, c'est l'essentiel !) Merci Quentin pour ta patience et ta pédagogie pendant nos cours, et pour nos échanges sur la musique et l'enseignement. Merci Esaïe de m'avoir accueilli dans ton bureau lors de mes passages à Paris. Merci Laurence et Hippolyte de m'accueillir chez vous lorsque j'ai besoin d'un pied-à-terre parisien. Merci Alexandre, Ulysse, Lucille et Marie pour nos après midi jeux de sociétés et soirées teppanyaki. Mes remerciements (et mes excuses) à toutes celles et tous ceux que j'ai oublié d'inclure ici...

Je veux enfin remercier chacun des membres de ma famille pour notre environnement familial, où je peux me ressourcer en votre compagnie. Merci Keja d'être là, c'est toujours un plaisir de discuter, de jouer, de réfléchir à des questions diverses et de plaisanter avec toi. Merci enfin Sandrine, pour notre vie commune et ton soutien sans faille depuis toute ces années.

Thank you all for the good moments, and here's to the future ones.

Contents

Résumé détaillé en français

Les problèmes étudiés dans cette thèses ont pour origine les tâches de traitement et d'analyse de texte, comme la validation de données, la compilation de code source, la classification de documents, l'extraction d'opinion ou encore la comparaison de corpus.

La recherche sur les techniques et algorithmes dans ce domaine a commencé dès les années 1950, et à donné lieu à la création de paradigmes très puissants (comme celui des expressions régulières [199]) et d'algorithmes à la fois élégants et efficaces (comme l'algorithme de recherche de motif de Knuth, Morris et Pratt [201]). Toutefois, ces outils et techniques ont été pensés pour résoudre les versions les plus communes mais aussi les plus basiques des problèmes d'analyse de texte, avec un objectif simple, et ne sont donc pas toujours adaptées aux tâches modernes, qui ont des objectifs plus complexe et qui comportent souvent une dimension *approximative*. Dans les tâche d'analyse de texte, la partie approximative peut venir de deux situations différentes : l'incertitude ou la relaxation des contraintes. Le premier cas correspond aux tâches dont certains aspect ne sont pas totalement spécifiés. Par exemple, on peut vouloir chercher dans un texte les occurrences d'un mot, en acceptant qu'il y ait une faute de frappe dans l'occurrence : la nature et la position de cette faute de frappe ne sont pas spécifiées. Dans d'autres cas, l'incertitude se situe dans les données, par exemple si une partie en a été perdue ou corrompue. Certaines situations combinent même les erreurs de mesure et l'incertitude sur les données : en bio-informatique [128], plus particulièrement en génomique, le domaine de l'analyse de séquences d'ADN, qui peuvent être représentées sous forme de texte (suite de lettre A,C,G et T). Dans ce domaine, il peut y avoir des erreurs lors du processus de séquence (extraction des données d'ADN), mais d'autre part, l'ADN de la plupart des êtres vivant subit des mutations spontanées lors de sa réplication. À cause de ces sources d'erreur, une recherche exacte est insuffisante pour déterminer par exemple si un gène donné apparaît dans le génome d'un patient. Pour pallier ce problème, les scientifiques ont développés des méthodes plus robustes, comme par exemple des algorithmes permettant d'identifier tous les fragment d'un texte qui sont (syntaxiquement) *similaires* à un motif donné [61, 107]. L'autre motivation pour les méthodes approximatives d'analyse de texte est le cas où l'on souhaitera avoir un résultat exact, mais que calculer un tel résultat demanderait trop de ressources de calcul, par exemple dans le cas de très grands volumes de données. Dans ce cas, une solution possible est d'utiliser des algorithmes plus frugaux en ressource, mais qui renvoient un résultat *approximatif*. Cela inclus les algorithmes qui ont une petite probabilité de donner une réponse incorrect, ou les algorithmes qui sont seulement capable d'identifier les instances positives et celles qui sont "clairement négatives". Ce dernier type d'algorithmes peut par exemple être utilisé comme procédure de filtrage, laissant seulement un petit nombre d'instances que l'on a pas pu déterminer à vérifier avec une procédure exact mais plus coûteuse.

Dans cette thèse, on étudie les algorithmes d'approximation pour deux aspects importants de l'analyse de texte : la recherche de motif (dans la Partie I) et l'appartenance à un

langage formel (dans la Partie II). Nous nous intéresseront à plusieurs problèmes algorithmique, et, pour chacun d'eux, notre but est de donner des algorithmes qui les résolvent le plus efficacement possible. Lorsque cela est important, nous implémentons l'algorithme, et rendons son code source disponible sous licence libre. Pour quantifier l'optimalité des résultats, on s'attache à donner des bornes inférieures qui s'approchent au plus de la complexité des algorithmes. Dans la plupart des cas, nos algorithmes et bornes inférieures sont basés sur et accompagnés de résultats nouveaux qui approfondissent notre compréhension d'objets combinatoires ou de phénomènes algorithmiques.

Recherche approximative de motif

> "Le motif P apparaît-il dans le texte T ?"

Cette question, simple en apparence et appelée le problème de recherche de motif (*pattern matching* en anglais), est centrale à de nombreuses tâches de traitement de données, de la validation à l'analyse. Ce problème est omniprésent : aujourd'hui, la majorité des communications internet passent par `HTTP` et `HTTPS`, des protocoles textuels de communication. De milliards de requêtes `HTTP` sont émises chaque jour, et la recherche de motifs est utilisée pour déchiffrer l'information contenue dans les *headers* de chaque requête. De même, les données émises par les applications web sont très souvent encodées en `JSON`, un format textuel permettant d'encoder des structures arbitrairement imbriquées. La recherche de motif permet d'extraire efficacement l'information de données sous forme `JSON` [162, 188].

Il existe de nombreuses variantes du problème de recherche de motif, que l'on obtient à partir des différentes définition possible de ce qu'est un *motif*, et de ce que *apparaître dans le texte* T veut dire. Par exemple, quand P est constitué de deux chaînes de caractères écarté d'un nombre fixé de positions, le problème est connu sous le nom de recherche avec écart [63] (en anglais, *gapped matching*). D'autre définitions fréquentes de "motif" inclus le cas d'une unique chaîne de caractère, ou d'un ensemble de caractères (c'est le problème de *dictionary matching* [16]). De plus, certaines variantes du problème demandent de seulement dire s'il existe une occurrence du motif, là où d'autres demandent de renvoyer l'information de la location de toutes les occurrences, ou seulement d'une d'entre elles.

Dans la première partie de la thèse, on se concentrera sur le cas basique (mais complexe) où P est une unique chaîne de caractères, et où l'on doit renvoyer l'information de toutes les occurrences *approximatives* de P dans le texte, pour différentes notions d'approximation. Ces tâches sont communément désignées sous le nom de recherche approximative de motif (en anglais, *approximate pattern matching*). La première notion d'approximation étudiée dans cette thèse est celle d'approximation *circulaire* : au lieu de chercher les occurrences P, on cherche les occurrences de *rotations* de P, c'est-à-dire les chaînes de caractères que l'on peut obtenir en écrivant P sur le périmètre d'un cercle et en coupant à n'importe quel point entre deux caractères. Ensuite, on s'intéresse à la recherche de motifs avec *jokers* (*wildcards* en anglais). Les jokers sont des caractères spéciaux qui sont considérés comme égaux de n'importe quel autre caractère, à la manière du " . " dans les moteurs d'expressions régulières ou de "`?`" dans les commandes `bash`. Un joker dans P correspond à une partie d'une requête qui n'est pas spécifiée. Enfin, on étudie également la recherche de motifs avec *erreurs*, où l'on doit trouver toutes les sous-chaîne de T qui sont *presque* égales à P, à quelques erreurs près. Ces différences peuvent correspondre à des erreurs de saisie, de mesure, ou à des mutations spontanées dans des brins d'ADN. Cette

notion de “presque égalité” est formellement définie à l’aide de distance entre chaînes de caractères.

Charalampopoulos, Kociumaka et Wellnitz [89] ont observé que de nombreux algorithmes récents pour la recherche (approximative) de motif peuvent s’énoncer en utilisant un ensemble restreint d’opérations sur les chaînes de caractères, indifféremment de l’implémentation de ces opérations. Dans cet ensemble d’opérations, qu’ils appellent le modèle `PILLAR`, les deux opérations les plus importantes sont la recherche interne de motif (en anglais, *internal pattern matching*, IPM) et le calcul de plus longue extension commune (*longest common extension*, LCE). Leur observation a deux conséquences. Premièrement, quel que soit le modèle de calcul (modèle RAM, ordinateur quantique, etc.), il suffit de donner une implémentation efficace de ces opérations, et on obtient immédiatement des algorithmes efficace pour la recherche approximative de motifs. Deuxièmement, si l’on énonce un algorithme dans le modèle `PILLAR`, alors on en obtient immédiatement une implémentation efficace dans tous les modèles où les opérations du modèles `PILLAR` ont une implémentation efficace. Par conséquent, la majorité des travaux compris dans la première partie de cette thèse sont basés sur le modèle `PILLAR` : on donne des implémentation efficaces des opérations `PILLAR` dans le modèles où elles manquent, et l’on énonce les algorithmes avec les opérations `PILLAR`.

Compromis temps-espace pour la recherche interne de motif.

Dans le Chapitre 4, nous donnons une structure de donnée efficace pour (une généralisation de) l’opération IPM, dans le modèle read-only. Cette structure de données offre un compromis entre temps-espace mémoire : on peut choisir quelle quantité d’espace mémoire la structure peut utiliser, et le temps de calcul pour répondre à une requête IPM dépendra de cette quantité de manière inversement proportionnelle ; plus l’espace mémoire alloué est grand, plus le temps de calcul sera faible. Avec le compromis espace-temps pour LCE donné par Kosolobov et Sivukhin [215], notre résultat permet d’obtenir une implémentation efficace de toutes les opérations `PILLAR` utilisant peu d’espace. Par conséquence, tous les algorithmes de recherche de motifs exprimés dans le modèle `PILLAR` peuvent être implémentés en utilisant le même compromis espace-temps. Nous utilisons également notre structure de données pour IPM pour donner des algorithmes utilisant un faible espace mémoire pour les problèmes de plus longue sous-chaîne commune et de recherche circulaire de motif, dans les modèle de streaming asymétrique et read-only.

Les résultats présentés dans ce chapitre ont fait l’objet d’une publication acceptée à CPM 2024 [51], co-écrite avec P. Charalampopoulos et T. Starikovskaya. Cet article a reçu le prix du meilleur article de la conférence !

Recherche de motifs avec différences et jokers.

Dans le Chapitre 5, on étudie le problème de recherche approximative de motif sous la distance de Hamming, dans les chaînes de caractères contenant des jokers. Ce chapitre contient deux résultats importants. Tout d’abord, nous donnons un algorithme efficace pour ce problème de recherche de motifs dans le modèle `PILLAR`. En utilisant les différentes implémentations du modèle `PILLAR`, nous montrons que cet algorithme donne des algorithmes efficaces pour le même problème dans plusieurs modèles de calcul. Ensuite, nous utilisons le premier résultat pour caractériser la structure des occurrences dans ce contexte, et termes de représentation avec un faible nombre de progressions arithmétiques, généralisant un résultat de Charalampopoulos et al. [89]. Ce premier résultat est

accompagné d'une borne inférieure comparable, obtenue en construisant un ensemble infini d'exemple, basés sur un résultat sur les ensembles sans progressions arithmétiques de Elkin [127].

Les résultats présentés dans ce chapitre ont fait l'objet d'une publication acceptée à ESA 2024 [52], co-écrite avec P. Charalampopoulos et T. Starikovskaya.

Plus longue extension commune avec jokers.

Dans le Chapitre 6, nous donnons une structure de données permettant de calculer les plus longues extensions communes dans les chaînes de caractères avec jokers (abrégé LCEW en anglais). Comme dans le Chapitre 4, notre structure de données présente un compromis esapace-temps : on peut choisir de réduire l'espace mémoire utilisé par la structure de données, mais cela augmente le temps de calcul utilisé pour répondre à une requête. Nous montrons ensuite comment cette structure de données peut être utilisée pour la recherche approximative de motifs et l'analyse de chaînes avec jokers. Enfin, nous exhibons une relation surprenante entre les LCE dans les chaînes avec jokers et la multiplication de matrice booléennes (parcimonieuses). Nous utilisation pour donner des bornes inférieures pour les structures de données combinatoires pour LCEW mais aussi un algorithme combinatoire et déterministe pour la multiplication de matrices booléennes parcimonieuses.

Les résultats présentés dans ce chapitre ont fait l'objet d'une publication acceptée à ESA 2024 [50], co-écrite avec P. Charalampopoulos et T. Starikovskaya.

Approximation de l'appartenance à un langage

Bien que la recherche de motif et ses variantes aient une grande variété d'application dans de nombreux domaines, elles ne peuvent qu'exprimer un nombre limité de propriétés des textes et chaînes de caractères. Le concept de *langage formel* permet de décrire des propriétés plus complexes : un langage formel est simplement un ensemble de chaînes de caractères. Étant donné une propriété sur les chaînes de caractères, on peut y associer un langage L en prenant l'ensemble des chaînes qui satisfont la propriété. Décider si une chaîne a une propriété est alors équivalent à décider si cette chaîne est dans le langage L associé. Ce point de vue est à la base de la théorie de la calculabilité, et, au fil des années, de nombreuses classes de langages formels ont été identifiées. Généralement, chaque classe a une définition syntaxique, et une expressivité limitée. Par exemple, les langage régulier sont définis par les expressions régulières et permettent de capturer les informations *séquentielles* dans un texte. Ils sont par exemple utilisée dans la validation de données de formulaires ou dans la conversion de code source en suite de lexèmes (une opération appelée "lexing" [19]). Les langages *algébriques*, plus expressifs, sont définis à l'aide de grammaires algébriques, et permettent de décrire des propriétés *hiérarchiques*. Par exemple, ils peuvent reconnaître structures imbriquées correctement, comme les expressions arithmétiques bien parenthésées, et ils sont utilisés lors du *parsing* de code source au cours de la compilation de programmes, c'est-à-dire l'étape de conversion d'une suite de lexèmes en un arbre de syntaxe abstrait [19].

À cause de son lien avec celle de décider une propriété, la question de décider l'appartenance à un langage formel a été très étudiée, et sa complexité algorithmique est pleinement comprise pour de nombreuses classes de langages. Par exemple, Schepper [269] donne une caractérisation "fine-grained" de la complexité de décider l'appartenance au langage d'une

expression régulière, en fonction de la structure de celle-ci. Pour l'appartenance à un langage algébrique, Cocke, Younger and Kasami [108, 192, 287] ont indépendamment découvert un algorithme avec une complexité cubique ; cet algorithme est connu sous le nom d'algorithme CYK. Quelques années plus tard, Valiant [281] a donné une réduction du problème d'appartenance à un langage régulier à celui de la multiplication de matrices booléennes. Plus récemment, Abboud et al. [11] ont montré que, sous l'hypothèse que les meilleurs algorithmes connus pour CLIQUE, l'algorithme de Valiant est optimal.

Dans ces deux exemples, on a des bornes inférieures (conditionnelles) qui montrent que l'algorithme pour décider l'appartenance à un langage de cette classe est optimal. Par conséquent, les projets de recherche plus récents se sont focalisés sur des variantes *approximatives* de l'appartenance à un langage formel.

Dans la deuxième partie de cette thèse, nous nous engageons dans la même démarche de recherche, et étudions la question d'approximer l'appartenance à un langage formel dans trois contextes différents, avec une attention particulière pour les algorithmes frugaux en espace mémoire.

La complexité de tester les langages réguliers

Tout d'abord, dans le Chapitre 8, on s'intéresse au *property testing*, le champ d'étude des procédures de décision frugales en information. Dans ce contexte, l'objectif est de décider si un mot d'entrée est dans un langage donnée, ou s'il en est *loin*, tout en utilisant aussi peu d'information que possible à propos de l'entrée. Suivant la direction du travail fondateur d'Alon, Krivelevich, Newman et Szegedy [24], on s'intéresse au cas des langages réguliers. Notre première contribution est de déterminer la complexité exacte du property testing des langages réguliers sous la distance de Hamming, en améliorant à la fois la borne supérieure et la borne inférieure de Alon et al. [24]. Ensuite, à partir de ce premier résultat, nous donnons une caractérisation complète de la complexité du problème : nous montrons qu'il existe trois classes de complexité pour ce problème, et donnons une caractérisation effective de ces classes.

Le travail présenté dans ce chapitre a fait l'objet de deux publications, l'une avec T. Starikovskaya, publiée à ICALP 2021 [48], l'autre avec C. Mascle et N. Fijalkow, accepté pour publication à ICALP 2025.

Calcul de la distance aux palindromes et aux carrés

Dans le Chapitre 9, on s'intéresse au problème de calculer la distance entre une chaîne de caractères et un langage, comme mesure de la proximité entre la chaîne et la propriété encodée dans le langage. On se place dans le *régime des distance faibles*, c'est-à-dire, le cas où l'on a un seuil de distance d, et l'on veut savoir si la distance est supérieure à d, ou si elle est inférieure ou égale, et, dans ce cas, quelle est la distance. Nous donnons des algorithmes *en ligne* et utilisant peu d'espace pour calculer la distance de Hamming et la distance d'édition entre une chaîne et les langages des palindromes ou des carrés. Ces résultats sont basés sur des procédures de filtrages qui utilisent des algorithmes de recherche approximative de motif. Dans deux cas, nous donnons des nouveaux algorithmes read-only de recherche approximative de motif adaptés à nos besoins.

Les résultats présentés dans ce chapitre ont fait l'objet d'une publication acceptée à ISAAC 2023 [49], co-écrite avec T. Kociumaka et T. Starikovskaya.

Algorithmes pour la longueur palindromique utilisant peut d'espace

Enfin, dans le Chapitre 10, on s'intéresse à une autre manière de quantifier la proximité d'une chaîne à un langage, qui permet de mieux capturer la sémantique du langage. Au lieu de mesurer la distance au langage, on compte le nombre minimum de morceaux en lequel il faut diviser la chaîne de telle sorte que chaque morceaux appartienne au langage. Dans le cas du langage des palindromes, ce nombre est appelé la *longueur palindromique* de la chaîne. Borozdin et al. [76] ont donné un algorithme qui calcule la longueur palindromique en temps et espace linéaire. Dans ce chapitre, nous donnons un algorithme utilisant peu d'espace qui calcule la longueur palindromique en temps presque-linéaire dans le régime des distances faibles. Pour obtenir cet algorithme, nous développons une nouvelle analyse de la structure des préfixes d'une chaîne qui ont longueur palindromique k (appelés *préfixes k-palindromiques*), et nous montrons que ces préfixes peuvent s'encoder de manière compacte en utilisant des objets combinatoire appelé les ensemble affines de préfixes. Nous montrons ensuite que cette structure peut être calculer efficacement en utilisant peu d'espace.

Les résultats présentés dans ce chapitre ont fait l'objet d'un article co-écrit avec J. Ellert et T. Starikovskaya, disponible en pré-publication sur ArXiv [53].

Chapter 1

General Introduction

The motivation for the work conducted during this thesis comes from problems that arise in text *analysis* and *processing* workloads, such as data validation, source code compilation, document classification, sentiment analysis, feature extraction, or corpora comparison.

Algorithms and techniques for text analysis have been studied since the 1950s, and research in this area has resulted in very powerful frameworks (such as regular expressions [199]) and elegant and efficient algorithms (such as the celebrated pattern matching algorithm of Knuth, Morris, and Pratt [201]). However, these tools and methods are designed to solve natural, common, but basic problems, and thus are not always well suited to modern text analysis tasks, which often have more complex objectives and often have an *approximate* component. The need for approximate techniques in text analysis arises from two different situations: *uncertainty* or *relaxation of the objective*. In the first case, some aspects of the task are not fully specified. For example, given a word, one can search for its occurrences that contain a typing error in a text. In other cases, the uncertainty may be in the *data*, for example, if some of the data has been lost or corrupted. Some situations combine both measurement error and data uncertainty: in computational genomics [128], the science of analyzing DNA data, errors may occur in the DNA extraction process, known as *sequencing*, and the DNA of most living organisms can spontaneously mutate. Because of such sources of error, *exact search* is insufficient to find out e.g. whether a gene is present in a patient's genome. To solve this problem, researchers have developed more robust methods, such as pattern matching algorithms that find all fragments of the text that are *similar* to a given pattern [61, 107]. The other motivation for approximate text analysis is when we want to get an exact result, but computing it is too computationally expensive, for example when dealing with a huge amount of data. In this case, the usual solution is to use algorithms that require less resources but may return an *approximate* answer. These include probabilistic algorithms that have a small probability of returning an incorrect answer, or algorithms that can only distinguish positive text instances from "obviously false" instances. The latter type of algorithms can be used as a filtering procedure, leaving only a small number of candidates to be verified with the more expensive exact procedures.

In this thesis, we study approximation algorithms for two important aspects of text analysis: pattern matching (in Part I) and language membership (in part Part II). Throughout this thesis, we consider several algorithmic problems. For each of them, our goal is to give algorithms that solve them as *efficiently* as possible. When relevant, we provide an open-source implementation of the algorithms. To quantify the degree of optimality of our results, we try to give *lower bounds* that are as close as possible to the

upper bound induced by the algorithm. In most cases, these algorithms and lower bounds are based on novel results that deepen our understanding of a combinatorial object or algorithmic phenomenon.

1.1 Approximate Pattern Matching

"Does the pattern P occur in the text T?"

This seemingly simple question, known as *pattern matching*, is at the heart of many data processing tasks, from validation to parsing. It is ubiquitous: most Internet traffic today relies on `HTTP` (or `HTTPS`), a text-based protocol. Trillions of `HTTP` requests are sent every day; pattern matching is needed to decipher the information contained in the headers of each such request. Similarly, data used in web applications is often encoded in `JSON`, a textual format that allows encoding arbitrary nested structures; pattern matching can be used to extract data from serialized `JSON` data [162, 188].

There are many variants of pattern matching, as one may vary the definition of what a *pattern* is and what *occurring in* T means. For instance, if P consists of two strings that are a fixed number of positions apart, then the problem is known as *gapped matching* [63]. Other common choices for P include a single string or sets of strings (dictionary matching [16]). In addition, some versions of the problem require reporting the existence of a match, while other versions require reporting information about the location of all matches, or only the leftmost or rightmost match in the text.

In this part of the thesis, we focus on the basic case where P is a single string, known as *pattern matching*, and one must report all *approximate occurrences* of P in the text, for various notions of "approximate occurrences". These tasks are commonly known as *approximate pattern matching*. The first notion of approximation studied in this thesis is *circular* approximation: instead of searching for occurrences of P, we search for *rotations* of P, i.e. strings that can be obtained by writing P along the circumference of a circle, and cutting at any position between two characters. Next, we study approximate pattern matching with *wildcards*. Wildcards are special characters that match any other character, similar to "`.`" in regular expression engines or "`?`" in shell commands. Placing a wildcard in P corresponds to a part of the query that is unspecified; for this reason, wildcards are sometimes called "don't cares" or "holes". Finally, we consider pattern matching with *errors* or *mismatches*, were the task is to find all substrings of T that are equal to P up to a few errors, that could arise typing errors on a keyboard, or from spontaneous mutations in DNA strands. This notion of approximate equality is formally defined later using *distances* over strings.

Charalampopoulos, Kociumaka, and Wellnitz [89] remarked that many recent algorithms for (approximate) pattern matching can be expressed in terms of a small set of string operations, independently of how these operations are implemented. In this setting, which they call the `PILLAR` model, the most crucial operations are *internal pattern matching* (IPM) and *longest common extension* (LCE) queries (these operations are formally defined in Chapter 3). Their observation has two consequences. First, in any framework, it suffices to implement these operations efficiently to unlock efficient pattern matching algorithms. Example of framework with efficient `PILLAR` implementations include the usual RAM model, algorithms on compressed strings, and the quantum computing model. Second, if an algorithm is stated in terms of the `PILLAR` operations, it can be efficiently implemented in all frameworks where the `PILLAR` operations are implemented. As a result, most of the work in this part of the thesis builds around the

PILLAR model: we provide efficient implementations of the crucial PILLAR operations in frameworks where they are missing, and express algorithms in terms of PILLAR operations.

1.1.1 Time-space trade-off for internal pattern matching.

In Chapter 4, we give an efficient data structure for (a generalization of) the IPM operation, in the read-only setting. This data structure offers a time-space trade-off: one can choose how much memory the data structure should use, and the time needed to answer IPM queries will depend on that quantity; the higher the memory allocated to the data structure, the lower the query time. This contribution is significant because, together with the time-space trade-off for LCE queries of Kosolobov and Sivukhin [215], it give a complete implementation of the PILLAR operations in small space. Therefore, any pattern matching algorithm implemented in the PILLAR model can be implemented with this time-space trade-off. We also use our IPM data structure to design space-efficient algorithms for the Longest Common Substring (LCS) and Circular Pattern Matching problems in small-space settings (streaming, asymmetric streaming, and read-only models).

The results presented in this chapter appeared in an article published at CPM'24 [51], co-authored with P. Charalampopoulos and T. Starikovskaya. This paper received the Best Paper Award of the conference.

1.1.2 Pattern Matching with mismatches and wildcards.

In Chapter 5, we study the problem of approximate pattern matching under the Hamming distance (also known as pattern matching with mismatches) in strings that may contain wildcards. Our contribution is twofold. First, we characterize the structure of occurrences in this setting, extending a result of Charalampopoulos et al. [89]. This result is completed by a comparable lower bound that we obtain by constructing an infinite family of examples, building on a combinatorial result on progression-free sets by Elkin [127]. Second, we use the first result to give an efficient algorithm for pattern matching with mismatches and wildcards. By leveraging the different implementations of the PILLAR model, this result implies efficient algorithms in several computation models.

The results presented in this chapter appeared in an article published at ESA'24 [52], co-authored with P. Charalampopoulos and T. Starikovskaya.

1.1.3 Longest Common Extension with Wildcards.

In Chapter 6, we give a data structure for answering LCE queries on strings with wildcards, called LCEW queries. As in Chapter 4, this result offers a time-space trade-off: the data structure can be made more compact, at the cost of an increased query time. As a result, we show that the above data structure yields efficient algorithms for approximate pattern matching and analysis of string with wildcards. Finally, we unveil a surprising connection between LCE queries in strings with wildcards and (sparse) Boolean matrix multiplication. As a result, we give a lower bound on the preprocessing-query time product of any data structure for LCEW, and a deterministic combinatorial algorithm for sparse Boolean matrix multiplication.

The results presented in this chapter appeared in an article published at ESA'24 [50], co-authored with P. Charalampopoulos and T. Starikovskaya.

1.2 Approximate Language Membership

While pattern matching and its variants have a wide variety of applications in many fields, they can only express a limited number of properties of strings. The paradigm of *formal languages* can be used to describe more complex properties: a formal language is simply a set of strings. Given a property P over strings, we can define a formal language L_P describing P by taking the set of all strings that have property P: deciding whether a string X has property P is then equivalent to deciding whether X is in L_P. This approach is at the root of the theory of computation, and, over the years, many classes of formal languages have been identified. Each class usually has a syntactic definition and a limited expressive power. For example, regular languages are defined by regular expressions and capture *sequential* information in the text: they are used, for example, in form data validation, or in converting source code into a sequence of tokens (an operation called "lexing" [19]). The more expressive *context-free* languages are defined by context-free grammars (CFGs) and capture *hierarchical* information: for example they can recognize well-nested structures, such as well-parenthesized arithmetic expressions, and are used when parsing source code into an *abstract syntax tree* during compilation [19].

Because of its connection to deciding properties, the question of deciding membership in a formal language has received a lot of attention, and its complexity is well understood for many classes of languages. For example, Schepper [269] gives a fine-grained characterization of the complexity of membership testing in the language of a regular expression, based on its structure. For membership in context-free languages, Cocke, Younger and Kasami [108, 192, 287] independently devised an algorithm that runs in cubic time – this algorithm is now colloquially known as the CYK algorithm. Later, Valiant [281] showed a reduction from context-free language membership to Boolean matrix multiplication, resulting in an algorithm that runs in time $O(BM(n))$ by exploiting fast matrix multiplication, where $BM(n)$ is the time complexity of multiplying two $n \times n$ Boolean matrices. More recently, Abboud et al. [11] showed that, under the assumption that the current best algorithms for Clique are optimal, Valiant's algorithm for parsing CFGs is optimal.

In both of the above examples, we now have (conditional) lower bounds that match the best upper bound for deciding membership in formal languages, so there is little hope of finding faster algorithms for this exact question. Therefore, subsequent research has focused on *approximate* versions of membership in formal languages. In the second part of this thesis, we follow this line of work, and study the question of approximating membership in a formal languages from three different points of view, with a particular emphasis on *space-efficient* algorithms.

1.2.1 The complexity of testing regular languages

First, in Chapter 8, we study *property testing*, a framework for *information-efficient approximate decision procedures*, where the task is to distinguish whether the input is in the language or if it is *far* from any word of the language, while requiring as little information about the input as possible. Following the seminal work of Alon, Krivelevich, Newman, and Szegedy [24], we focus on the case of *regular languages*. Our first contribution is to determine the exact complexity of this problem in the case of the Hamming distance, improving both the upper and lower bounds of Alon et al. [24]. Building on this first result, we then give a complete characterization of the complexity property testing of regular languages under the Hamming distance: we identify all complexity classes for this problem, and give an effective combinatorial characterization in each class.

The work presented in this chapter previously appeared in two articles: one with T. Starikovskaya [48], which was published at ICALP'21, and one with C. Mascle and N. Fijalkow, accepted for publication at ICALP'25.

1.2.2 Computing the distance to palindromes and squares, online and in small-space

In Chapter 9, we study the problem of computing the *distance* of a string to a language, as a measure of how close the string is from having the property defined by the language. This chapter is focused on the *low-distance regime*, that is, the case where we have a distance threshold d, and want to either know that the input is at distance more than d from the language, or that it is at distance less than d, and in this case we require the exact distance. We give *small-space online algorithms* for computing the Hamming or edit distance of a string to the languages of *palindromes* and the language of *squares*, in the low-distance regime. These results are based on filtering procedures that leverage approximate pattern matching algorithms (similar to the ones described in Section 1.1). In two cases, we give a new read-only algorithm for this task, tailored to our needs.

The results presented in this chapter appeared in an article published at ISAAC'23 [49], co-authored with T. Kociumaka and T. Starikovskaya.

1.2.3 Small-space algorithms for palindromic length

Finally, in Chapter 10, we study another approach to quantifying the proximity of a string to a language, which better captures the semantics of the language. Instead of measuring the distance to the language, we count the minimum number of pieces that the string must be split into so that each piece is in the language. For the case of the language of palindromes, this value is called the *palindromic length* of the string. Borozdin, Kosolobov, Rubinchik, and Shur [76] gave an algorithm that computes the palindromic length in linear time and linear space. In this chapter, we give a small-space algorithm for computing the palindromic length in near-linear time in the low-distance regime. To obtain this algorithm, we develop a novel analysis of the structure of the prefixes of a string that can be decomposed into a given number k of palindromes (called *k-palindromic prefixes*), and show that theses prefixes can be compactly encoded using a small number of combinatorial objects called *affine prefix sets*. We then show that the structure underlying this analysis can be computed efficiently in a space-efficient manner.

The results presented in this chapter resulted in an article co-authored with J. Ellert and T. Starikovskaya, available as a preprint on the ArXiv [53].

Chapter 2

Preliminaries

This section introduces basic concepts that we assume familiarity with in this thesis.

2.1 Strings

Strings are finite, ordered sequences of elements drawn from a finite set, called the *alphabet*, usually denoted Σ. The elements of Σ are called *letters* or *characters*. For an integer $n \geq 0$, we denote the set of all strings of length n by Σ^n, and we set $\Sigma^{\leq n} = \bigcup_{m=0}^{n} \Sigma^m$ as well as $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$. The (unique) empty string is denoted by ε. For integers $i, j \in \mathbb{Z}$, denote $[i..j] = \{k \in \mathbb{Z} : i \leq k \leq j\}$, and $[i..j) = \{k \in \mathbb{Z} : i \leq k < j\}$. For a string $T \in \Sigma^n$ and an index $i \in [1..n]$, the i-th letter of T is denoted by $T[i]$. We use $|T| = n$ to denote the length of T. For two strings S, T, we use either ST or $S \cdot T$ to denote their concatenation $S[1] \cdots S[|S|]T[1] \cdots T[|T|]$. For an integer $m \geq 0$, the string obtained by concatenating S with itself m times is denoted by S^m, with the convention that $S^0 = \varepsilon$. A string S is a *square* if there exists a string T such that $S = T^2$. In a slight abuse of notation, we denote by S^∞ the string obtained by concatenating infinitely many copies of S.

Substrings, prefixes and suffixes. Let T be a string of length n. For integers i, j, $T[i..j]$ denotes the *substring* or *fragment* $T[i]T[i+1] \cdots T[j]$ of T if $1 \leq i \leq j \leq n$ and the empty string ε otherwise. We extend this notation with $T[i..j) = T[i..j-1]$ and $T(i..j] = T[i+1..j]$. When $i = 1$ or $j = n$, we may omit these indices, i.e., we write $T[..j]$ for $T[1..j]$ and $T[i..]$ for $T[i..n]$. We say that a string P is a *prefix* of T if there exists $j \in [1..n]$ such that $P = T[..j]$, and a *suffix* of T if there exists $i \in [1..n]$ such that $P = T[i..]$. A substring (hence also a suffix or prefix) of T is *proper* if it is shorter than T.

Rotations, periodicity, and primitivity. We define the *cyclic left rotation* (or rotation, for short) of T $\mathsf{rot}(T) = T[2] \cdots T[n]T[1]$. In general, a rotation $\mathsf{rot}^s(T)$ with shift $s \in \mathbb{Z}$ is obtained by iterating $|s|$ times the operation rot if $s > 0$, or its inverse rot^{-1} (the cyclic right rotation) if $s < 0$. We say that a string S is a rotation of T if there exists an integer s such that $S = \mathsf{rot}^s(T)$. A non-empty string T of length n is *primitive* if it is distinct from its non-trivial rotations, i.e., if $T = \mathsf{rot}^s(T)$ holds only if n divides s. Equivalently, T is *primitive* if $S^m = T$ implies that $m = 1$ and $S = T$.

A positive integer ρ is a *period* of a string T of length n if $T[i] = T[i+\rho]$ for all $i \in [1..n-\rho]$. A fundamental tool for analyzing periodicities is the celebrated Periodicity

Lemma by Fine and Wilf [134], which is stated below.

Fact 2.1.1 (Periodicity Lemma [134]). *If p and q are distinct periods of a string of length at least $p+q-\gcd(p,q)$, then $\gcd(p,q)$ is a period of the string.*

The smallest period of T is called *the* period of T and is denoted by $\mathsf{per}(T)$. If $\mathsf{per}(T) \leq n/2$, then T is called *periodic*.

Fact 2.1.2 ([115]). *For any string T, the prefix $T[..\mathsf{per}(T)]$ is primitive.*

Reversal and palindromes. We use T^R or $\mathrm{rev}(T)$ to denote the reverse of T, that is, the string $T^R = T[n]T[n-1]\cdots T[1]$. A string T is a *palindrome* if $T^R = T$.

Occurrences and pattern matching. A fragment $T[i..j]$ of a string T is called an *occurrence* of a string P if $T[i..j] = P$; in this case, we say that P *occurs* at position i of T. In some chapters of this thesis, we use "occurrences" to refer to the starting position i or the ending position j of the occurrence $T[i..j]$; this will be explicitly stated in the corresponding chapters. In the *pattern matching* task, we are given a text T and a pattern P, and must report all occurrences of P in the text T.

Longest common prefix and longest common extension. Given two strings S and T, the *longest common prefix* of S and T, denoted by $\mathsf{LCP}(S,T)$, is the length of the longest prefix of S that is also a prefix of T. Given two indices $i \in [1..|S|]$ and $j \in [1..|T|]$, the *longest common extension* (LCE) of S and T at indices i, j, denoted by $\mathsf{LCE}_{S,T}(i,j)$, is equal to the longest common prefix of $S[i..]$ and $T[j..]$. When S and T are clear from the context, we may omit them and just write $\mathsf{LCE}(i,j)$.

Metrics on string: Hamming and edit distances. Let S and T be two strings. The Hamming distance between S and T, denoted $\mathsf{hd}(S,T)$, is the number of indices at which S and T differ, if they have the same length, and $+\infty$ otherwise. Formally:

$$\mathsf{hd}(S,T) = \begin{cases} \sum_{i=1}^{|S|} \mathbb{1}_{S[i]\neq T[i]} & \text{if } |S| = |T|, \\ +\infty & \text{otherwise.} \end{cases}$$

The edit distance is defined using three elementary operations on strings:

- the *insertion* of a character $a \in \Sigma$ at position $i \in \{0,\ldots,|S|\}$:

$$\mathsf{ins}(S,i,a) = S[1..i]aS[i+1..]$$

- the *substitution* with a character $a \in \Sigma$ at position $i \in \{1,\ldots,|S|\}$:

$$\mathsf{sub}(S,i,a) = S[1..i-1]aS[i+1..]$$

- the *deletion* of a character at position $i \in \{1,\ldots,|S|\}$:

$$\mathsf{del}(S,i) = S[1..i-1]S[i+1..]$$

The edit distance between S and T is the minimum number of elementary operations that are needed to obtain T starting from the string S.

Using this framework, the Hamming distance between S and T can be reformulated as the minimum number of substitutions that are needed to obtain T from S.

2.2 Formal Languages

A formal language is a (potentially infinite) set L of strings, that is, a subset of Σ^*. In this thesis, we simply use *language* for formal languages, for conciseness. Let L, L' be languages. Extending the notations used for strings, we define the concatenation of languages, denoted $L \cdot L'$ or LL', as $L \cdot L' = \{w \cdot w' : w \in L \text{ and } w' \in L'\}$. Similarly, we define language exponentiation as follows: for $m \geq 0$, the language L^m is defined as $\{\varepsilon\}$ if $m = 0$, and $L \cdot L^{m-1}$ otherwise. By unrolling the recursion, we obtain an alternative definition:

$$L^m = \begin{cases} \{\varepsilon\} & \text{if } m = 0 \\ \{w_1 w_2 \dots w_m : \forall i, w_i \in L\} & \text{otherwise.} \end{cases}$$

Intuitively, exponentiation corresponds to iterating concatenation a fixed number of times. The *Kleene star* of L, denoted L^*, corresponds to repeating concatenation for an arbitrary number of iterations. Formally,

$$L^* = \bigcup_{m \geq 0} L^m = \{\varepsilon\} \cup \{w : \exists m \geq 1 \text{ and } w_1, \dots, w_m \in L \text{ s.t. } w = w_1 \dots w_m\}.$$

2.3 Complexity and Models of Computation

2.3.1 Big-O notation.

In this thesis, we use the "big-O" notation of Landau [222] to describe the asymptotic behavior of functions, such as the computational complexity of algorithms. We recall here the formal definition of this family of notations (see [200] for a reference and a historical exposition). We start with the $O(\cdot)$ notation, that denotes the relation of asymptotic domination. Let $f, g : \mathbb{N} \mapsto \mathbb{R}_{>0}$ be positive functions. They satisfy $f(n) = O(g(n))$ if there exists a constant $C > 0$ such that for all sufficiently large n, we have $f(n) \leq C \cdot g(n)$. For example, $f(n) = O(1)$ means that f is upper bounded by a constant that is independent of n. The $\Omega(\cdot)$ ("big-omega") notation denotes the reciprocal relation, that is:

$$f(n) = \Omega(g(n)) \iff g(n) = O(f(n)).$$

The $\Theta(\cdot)$ ("theta") notation denotes the "symmetric" version of $O(\cdot)$, i.e.

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } g(n) = O(f(n)).$$

The $o(\cdot)$ ("little-O") notation denotes the "strict" version of $O(\cdot)$: we have $f(n) = o(g(n))$ if and only if for *any* constant $\varepsilon > 0$, we have $f(n) \leq \varepsilon g(n)$ for all sufficiently large n. The $\omega(\cdot)$ ("little-omega") notation denotes the inverse relation of $o(\cdot)$. Additionally, we use the $\tilde{O}(\cdot)$ notation to hide factors polynomial in the logarithm of function:

$$f(n) = \tilde{O}(g(n)) \iff \exists c > 0 : f(n) = O(g(n) \log^c g(n)).$$

2.3.2 Computational complexity

Throughout this thesis, we assume the standard word Random Access Memory (word-RAM, or simply RAM) model of computation [241], using machine words of width $w = \Theta(\log n)$ when processing an input string of length n. In this model, the algorithm may perform basic operations (such as comparisons, arithmetic, or copy) on integers encoded

on $\Theta(\log n)$ bits in a single step. Therefore, if the input strings are encoded over an integer alphabet of size polynomial in n, characters can be stored in a single machine word, and operations on characters take unit time. Furthermore, accessing (reading or writing) the value stored in the memory at location given by an address stored on a machine word is also an elementary operation that takes a single step (hence the name *random access*).

In this model, the most widely used measures of computational complexity are the (worst-case) *time complexity* and the *space complexity*. The time complexity of an algorithm A is a function that maps n to the maximum, over all inputs x of length n, of the number of basic RAM operations that A on x. The space complexity instead counts the number of memory locations that the algorithm writes to, under the assumption that the algorithm never reads a memory before writing to it. We may also report the space complexity as a number of *bits of space* that the algorithm uses. In this case, the two quantities are within a logarithmic factor of one another. Let $s(n)$ denote the space complexity in number of words, and $s_b(n)$ the space complexity in number of bits used. As each bit that the algorithm uses belongs to some memory word, we have $s(n) \leq s_b(n)$. Furthermore, as each machine word uses $w = \Theta(\log n)$ bits, we have $s_b(n) \leq w \cdot s(n) = O(s(n) \log n)$, which implies that $s_b(n) = \tilde{O}(s(n))$.

Amortized time complexity. There are data structures that can perform an operation very efficiently in most cases, and must perform a time-consuming computation in the remaining few cases. Therefore, the worst-case time complexity of the operation is that of the expensive computation, but if that case is sufficiently rare, an algorithm that runs the operation many times will be only marginally slowed down. This idea is formalized as *amortized complexity*. An operation has amortized complexity $O(f(n))$ if, for any integer k, applying that operation k times to any given input takes time $O(k \cdot f(n))$.

2.3.3 Input access models.

In some cases, we place restrictions on how some parts of memory (usually the input) can be accessed.

Read-only model. In the *read-only* model, the algorithm has *read-only* random access to the input, i.e. the algorithm can read from the memory where the input is stored, but not cannot write to it. Therefore, the space occupied by the input is never included in the space complexity.

Online algorithms. In this thesis, *online algorithms* refers to string algorithms that acquire information about the input sequentially at runtime. After performing its computation on the current input, the algorithm receives the next character of the input and must update its result accordingly. The main measure of complexity of online algorithms is the *time per character*, which measures the time it takes to update the result after receiving a new character.

Streaming model. In the *streaming* model, the algorithm has *sequential* access to the input string T. In this model, the input is stored in a special read-only memory location m such that the i-th time the algorithm reads from m, it receives the i-th character $T[i]$ of the input (or $T[n]$ if $i > n$). Another read-only memory location stores the length n of T. In this model, after receiving $T[i]$, the algorithm cannot access information about $T[j]$

for $j < i$ unless it is explicitly stored in the writable part of the memory – in which case it would be included in the space complexity. Algorithms in the streaming model are a restricted form of online algorithms: they access the input in a sequential manner, but have no memory of previously seen parts of the input.

2.4 Tries and the Suffix Tree

The suffix tree is a classical but important data structure for string processing (see e.g. [284]). In this section, we recall its definition and important properties, and give an overview of existing variants of the suffix tree.

2.4.1 Tries and compressed tries.

First, we recall the notion of *tries*, a kind of search tree adapted to string processing. Given a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of strings over an alphabet Σ, a *trie* for $\mathcal{S}$ is a rooted tree whose edges are labeled with letters from Σ, and whose internal nodes correspond to prefixes of the S_i's. A trie additionally satisfies the following properties:

1. For each character $a \in \Sigma$, each internal node has at most one child edge labeled a.
2. For every $S_i \in \mathcal{S}$ and for every $0 \leq j < |S_i|$, the node corresponding to $S_i[..j]$ is the parent of $S_i[..j+1]$, and the edge between them is labeled with the character $S_i[j+1]$.

An internal node of the tree corresponds to the string obtained by concatenating the labels of the edges on the path from the root to that node. The first property above ensures that paths corresponding to common prefixes of strings in $\mathcal{S}$ are merged in the trie, see Fig. 2.1 for an illustration.

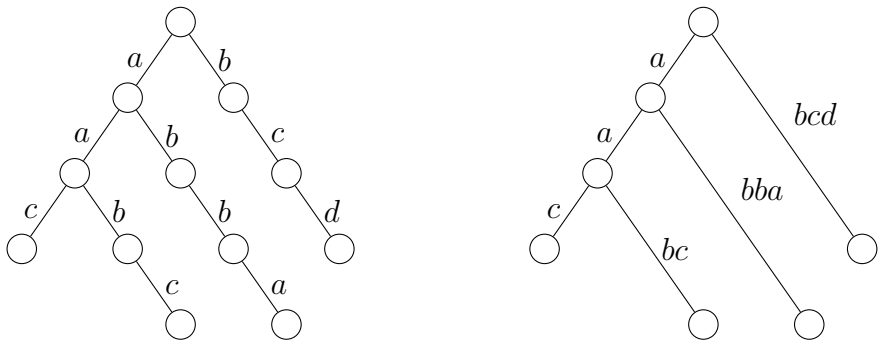

Figure 2.1: The trie for the set of strings $\mathcal{S} = \{aac, aabc, abba, bcd\}$ (left) and the corresponding compressed trie (right).

Compressed tries. In the absence of branching, long paths from $S[..i]$ to $S[..j]$ can be compressed and replaced by a single edge labeled by $S[i+1..j]$ (see Fig. 2.1). This simplification ensures that a compressed trie for a set $\mathcal{S}$ of n strings contains $O(n)$ internal nodes, regardless of the length of the strings [168]. In what follows, we will simply use "trie" to refer to compressed tries.

2.4.2 The Suffix Tree

Originally introduced by Weiner [284], the suffix tree T_S of a string S is a (compressed) trie of all suffixes of S. Naive construction algorithms for T_S take $O(n^2)$ time, while Weiner's algorithm runs in linear time for constant-size alphabet. Weiner's work was later extended by Ukkonen [280], who gave an algorithm for constructing the suffix tree of a string *on-line*, and by Farach [130], who gave a linear time algorithm for constructing the suffix tree of strings over alphabets that can be sorted in linear time.

Applications of the Suffix Tree

The suffix tree can be used as an indexing data structure to efficiently solve string processing problems. Namely, after linear-time preprocessing of the suffix tree of S, we can perform the following operations [168]:

- Given a pattern P, find an arbitrary occurrence or count the number of occurrences of P in S, in time $O(|P|)$.
- Given two indices i, j, compute the longest common extension $\mathsf{LCE}_{S,S}(i, j)$ of S at positions i and j in *constant time*.

Numerous combinatorial string problems, such as Longest Palindromic Substring, Longest Common Substring, computing the LZ-Decomposition and counting the number of unique substrings have a linear-time algorithm based on the suffix tree [168].

Sparse Suffix Trees.

In some applications (see for example Chapter 4), we need a trie that contains only a given subset E of b suffixes of S. Such a trie is called a *sparse suffix tree*. To build such a sparse suffix tree, one can build a complete suffix tree and remove the nodes corresponding to suffixes not in E. This algorithm uses $\Theta(n)$ space in the worst case, since it builds the whole suffix tree. However, a compressed trie containing b strings has $O(b)$ nodes and thus uses $O(b)$ space, so the above algorithm is not space-optimal. On the other hand, the algorithm that inserts all b suffixes into a trie one by one takes $O(nb)$ time and $O(b)$ space: its space usage is optimal but the algorithm is much slower. There are algorithms that achieve both near-optimal time and space complexity [69, 157, 215]: the current state of the art is by Kosolobov and Sivukhin [215], who gave a deterministic algorithm that computes sparse suffix trees in $O(n \log_b n)$ time and $O(b)$ space (for $b \geq \log^2 n$).

Part I

Approximate Pattern Matching

Chapter 3

Introduction and Overview

In Section 3.1, we present the different notions of approximate string matching studied in this thesis. In Section 3.2, we recall results related to periodicity in pattern matching, which will be a recurring theme in the next chapters. In Section 3.3, we present the `PILLAR` model, a powerful abstract framework for creating model-independent string algorithms introduced by Charalampopoulos, Kociumaka, and Wellnitz [89], on top of which multiple results of this thesis are built. Finally, in Section 3.4, we give an overview of the contents of the chapters of the first part of this thesis.

3.1 Approximate String Matching

3.1.1 Exact string matching

Before enumerating the variants of approximate string matching that we study in this thesis, we give a high-level overview of the algorithms and data structures for *exact* string matching.

In exact string matching, the goal is to find all exact occurrences of a pattern P of length m in a text T of length n, that is, all substrings of T that are equal to P. Because of its importance, this problem has received a lot of attention between the 60's and the 80's. The most influential exact string matching algorithms from this period are the deterministic, linear-time algorithm of Knuth, Morris, and Pratt [201] and the randomized algorithm of Karp and Rabin [190], which uses constant space in the read-only model. More recently, a line of work initiated by Porat and Porat [250] and later improved by Breslauer and Galil [77] resulted in a $O(\log m)$-space streaming algorithm that uses constant time per character and reports all exact occurrences of P in T. Classical data structures such as the *suffix tree* (see Section 2.4) or *suffix array* allow to efficiently solve multiple instances of pattern matching against the same text. After an indexing phase that takes $O(n)$ time, these data structures allow answering pattern matching queries in time close to $O(m)$ (with additional time linear in the number of occurrences if reporting is required). We will use some of these data structures and algorithms as subroutines in the algorithms presented in the next chapters.

3.1.2 Approximate pattern matching

A natural extension of exact string matching is *approximate string matching*, where the goal is to find all substrings of T that are *similar* to P. This variant allows modeling

uncertainty and arises, for example, in problems related to genomics, where mutations (substitution, insertion or removal of a character) can occur spontaneously.

Similarity between strings is best captured by a notion of distance: the most widely used are the Hamming distance and the edit distance (also known as Levenshtein distance). The corresponding problems are known as pattern matching with *mismatches* and pattern matching with *edits* (or *errors*). For the Hamming distance, Abrahamson [14] and Kosaraju [213] independently developed an $O(n\sqrt{m \log m})$-time algorithm that computes the Hamming distance between the pattern and each m-length substring of the text using convolutions via the Fast Fourier Transform (FFT). This result was recently improved by Chan et al. [86], who gave a randomized algorithm that runs in time $O(n\sqrt{m})$, and by Jin and Xu [187] who gave a deterministic algorithm running in time $O(n\sqrt{m \log \log m})$.

In this thesis, we study a variant of approximate pattern matching parameterized by an integer k: the goal is to identify all substrings of T whose (Hamming or edit) distance to P is at most k.

Example 3.1.1. Let $P = \texttt{abcad}$, and consider the case of the Hamming distance with $k = 1$. In the string $T = \texttt{cbd}\underline{\texttt{adcad}}\texttt{bb}\underline{\texttt{abcad}}$, there are two 1-mismatch occurrences of P, starting at positions 4 and 11 (underlined) (the latter is also an exact occurrence).

For the Hamming distance, the state-of-the-art solutions are the $O(n\sqrt{k \log k})$-time algorithm of Amir et al. [32], the $O(n + (n/m) \cdot k^2)$-time algorithm of Chan et al. [85], and the $O(n + kn/\sqrt{m})$-time algorithm of Chan et al. [86], which provides a smooth trade-off between the two previous solutions and improves the bound for some range of parameters. The last two algorithms are randomized, and deterministic alternatives (at the expense of additional polylogarithmic factors) have been presented in [89, 105, 158]. For the edit distance, Cole and Hariharan [112] gave an algorithm that runs in time $O(n + (n/m) \cdot k^4)$. Their result was recently improved by Charalampopoulos et al. [94], who gave an algorithm that runs in time $O(n + (n/m) \cdot k^{3.5})$.

As for exact pattern matching, the approximate versions have also been studied in the streaming model [164, 207, 250, 255]. Clifford et al. [107] gave a remarkable construction that draws on error-correcting code theory to extend the sketch-based approach of Karp and Rabin [190] into an efficient streaming algorithm for pattern matching with mismatches that uses $\tilde{O}(\sqrt{k})$ time per character and $\tilde{O}(k)$ space. Building on their work, Bhattacharya and Koucký [61] gave a streaming algorithm for the edit distance case.

In this thesis, we will use some of the above algorithms as subroutines for more complex tasks, such as computing distance to languages in Chapter 9.

3.1.3 Pattern matching with partial data: wildcards

Using mismatches or errors as presented above models *general* uncertainty in the input strings. In some cases, the uncertainty is *localized*: for example, a corrupted part of the data may be missing, or part of the pattern may be irrelevant for the query. In this case, we can use a more adaptive approach and place a *wildcard*[1] $\lozenge \notin \Sigma$, a special symbol that matches any character in $\Sigma \cup \{\lozenge\}$, in each of these positions, and then performing exact pattern matching.

Example 3.1.2. Let $P = \texttt{ab}\lozenge\texttt{a}$. In the string $T = \texttt{c}\lozenge\underline{\texttt{da}\lozenge\texttt{cad}}\lozenge\texttt{ac}$, there is an occurrence of P starting at position 4 (underlined), as the wildcard $\lozenge$ in P matches $\texttt{c}$, and the one in T matches $\texttt{b}$.

[1] Wildcards are also known in the literature as "don't cares" or "holes".

Already in 1974, Fischer and Paterson [140] presented an $O(n \log m \log \sigma)$-time algorithm for pattern matching with wildcards (where σ is the size of the alphabet Σ). Subsequent works by Indyk [184], Kalai [189], and Cole and Hariharan [111] culminated in an $O(n \log m)$-time deterministic algorithm. A few years later, Clifford and Clifford [100] presented a very elegant algorithm with the same complexities. All the above solutions are based on fast convolutions via the FFT. Golan et al. [163] gave a streaming algorithm for the problem that uses $O(D + \log m)$ time per character and $O(D \log m)$ space for patterns with D wildcards.

Unsurprisingly, the *approximate* pattern matching problem in strings with wildcards has also received significant attention. Conceptually, it covers the case when some of the corrupt positions are known, but not all of them. Notable results for these problems include the $\tilde{O}(nk)$-time algorithm of Clifford et al. [102] for pattern matching with k mismatches and wildcards, and the $\tilde{O}(n\sqrt{km})$-time algorithm of Akutsu [20] for pattern matching with k errors and wildcards. Chapter 5 contains for a more complete overview of results in this line of research.

In Chapter 5, we study the problem of pattern matching with mismatches and wildcards and give an efficient algorithm for strings that contain few contiguous groups of wildcards. In Chapter 6, we give a data structure for LCE queries in strings with wildcards. As an application, we obtain a more efficient algorithm for pattern matching with edits and wildcards.

3.1.4 Circular pattern matching

Another notion of approximate pattern matching is that of *circular* pattern matching, where the goal is to find occurrences of *rotations* of the pattern in the text.

Example 3.1.3. Let $P = \mathtt{abca}$. In the string $T = \mathtt{abda\underline{caab}dbac}$, there is a circular occurrence of P starting at position 5 (underlined), as $\mathsf{rot}^2(P) = \mathtt{caab}$.

The interest in occurrences of rotations of a given pattern is motivated by applications in Bioinformatics and Image Processing: in Bioinformatics, the starting position of a biological sequence can vary significantly due to the arbitrary nature of sequencing in circular molecular structures or inconsistencies arising from different standards of linearization applied to sequence databases; and in Image Processing, the contour of a shape can be represented using a directional chain code, which can be viewed as a circular sequence, particularly when the orientation of the image is irrelevant [39].

For strings over an alphabet of size σ, the classical read-only solution for this problem, based on the suffix automaton of $P \cdot P$, runs in $O(n \log \sigma)$ time and uses $O(m)$ extra space [230]. Recently, Charalampopoulos et al. [93] showed a simple $O(n)$ time and $O(m)$ extra space solution. The problem has been also studied from the practical point of view [97, 143, 275] and for indexing data structures that allow to perform circular matching of multiple patterns against a single text. For example, Hon et al. [177] gave a space-efficient algorithm for computing the circular suffix tree of a string, a data structure that generalizes the suffix tree to circular pattern matching. Other results include succinct indexes [176] and data structures for dictionary matching [182]. Finally, recent work has considered the problem of *approximate* circular pattern matching, under both the Hamming [92, 93] and the edit distance [96].

In Chapter 4, we study small-space data structures for *internal* pattern matching and, as an application, give a small-space algorithm for exact circular pattern matching.

3.2 Periodicity and the Structure of Occurrences

The notion of *periodicity* is intimately connected to pattern matching, and more precisely to the structure of occurrences of a pattern in a text.

3.2.1 A standard trick in pattern matching

Before giving the details of this connection, we discuss a technique[2] introduced by Clifford and Clifford [100] that allows reducing the problem of pattern matching to the case where the text is not much longer than the pattern. More precisely, pattern matching between a text T of length n and a pattern P of length m can be reduced to $O(n/m)$ instances of pattern matching between P and texts $T_1, T_2, \ldots$ of length $3m/2$ (except the last, which may have length less than $3m/2$). The text T_1 consists of the first $3m/2$ characters of T, and each subsequent T_i is obtained by taking the substring of length $3m/2$ of T to the right of T_{i-1} that overlaps it by $m-1$ positions. Then, every occurrence of P in T is contained in exactly one of the T_i's. Since we cover at least $m/2$ new characters of T with each new T_i, there are $O(n/m)$ T_i's. Thus, an algorithm with runtime $t(m)$ for the case $|T| \leq 3|P|/2$ readily implies an algorithm for the general case that runs in time $O(t(m) \cdot n/m)$, since one can run $O(n/m)$ separate instances and aggregate the results.

3.2.2 The structure of occurrences

Let us now discuss *why* the assumption that $n \leq 3m/2$ is convenient: in this case, the occurrences of P in T can be represented succintly using a single *arithmetic progression* (see Fact 3.2.1, and Fig. 3.1 for an illustration). A set $\mathcal{S}$ of integers forms an arithmetic progression if it contains evenly spaced integers, or, more formally, if there exist integers a, b and t such that $\mathcal{S} = \{a + i \cdot b, i = 0, \ldots, t\}$. The integer b is called the *difference* of the arithmetic progression.

Fact 3.2.1 ([134, Corollary of Theorem 1]). *Let P be a pattern of length m and let T be a text of length $n \leq 3m/2$. At least one of the following holds:*

- *("aperiodic case") there is at most one occurrence of P in T, or*
- *("periodic case") P is periodic, and the occurrences of P in T form an arithmetic progression with difference $\mathsf{per}(P)$.*

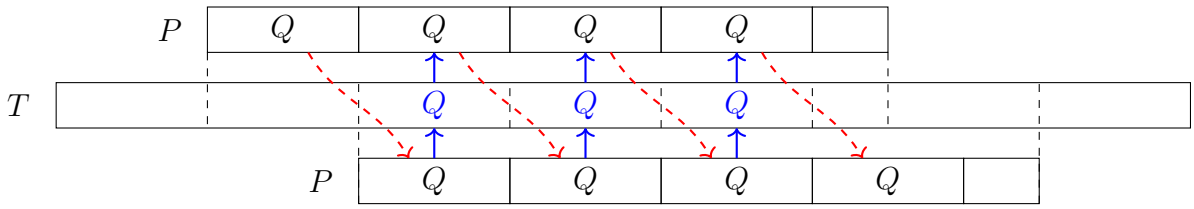

Figure 3.1: Illustration of the periodicity induced by multiple occurrences of P in a short text. Because $|T| \leq 3|P|/2$, two occurrences of P in T will have a large overlap, of at least $|P|/2$ characters. The prefix Q of P that spans this overlap is therefore repeated in P. Red dashed arrows represent equality between the same substring of P, and solid blue arrows represents equality between matching parts of the text and of the pattern.

[2] In the literature, this technique is literally known as "the standard trick" of pattern matching.

As an application of the standard trick, we can derive the following result for texts of arbitrary lengths.

Corollary 3.2.2. *Let P, T be strings. The occurrences of P in T can be partitioned into $O(|T|/|P|)$ arithmetic progressions with difference $\mathsf{per}(P)$.*

Fact 3.2.1 is a powerful and widely used tool for designing pattern matching algorithms. The case where P is aperiodic is easy, since there is at most one occurrence to find. In the periodic case, one can exploit the periodicity to make the algorithm more efficient. The streaming pattern matching algorithm of Porat and Porat [250] is an illustration of this periodic/aperiodic case distinction.

Another notable consequence of Fact 3.2.1 is that, when $n \leq 3m/2$, the occurrences of P in T have a compressed representation: using the three numbers (a, b, t) that describe the arithmetic progression, we can represent these occurrences using $O(1)$ space. This allows for very efficient data structures that can perform internal pattern matching (defined in Section 3.3) in *constant time*, even when m is large.

3.2.3 The structure of approximate occurrences

An important part of the work in this thesis focuses on *approximate* pattern matching, which cannot directly take advantage of the properties of *exact* pattern matching. However, Charalampopoulos, Kociumaka, and Wellnitz [89] gave an extension of these structural results for approximate pattern matching under the Hamming and edit distances. Because they consider *approximate* occurrences, the results consider a weaker notion of periodicity. In the case of the Hamming distance, their result is the following (the case of the edit distance is similar).

Fact 3.2.3 ([89, Theorem 3.1])**.** *Let P be a pattern of length m and T be a text of length $n \leq 3m/2$. At least one of the following holds:*

- *T contains $O(k)$ k-mismatch occurrences of P, or*
- *P is approximately periodic: there exists a string Q of length $O(m/k)$ such that $\mathsf{hd}(P, Q^\infty) < 2k$, and the k-mismatch occurrences of P in T can be partitioned into $O(k)$ arithmetic progressions with difference $|Q|$.*

Here, "k-mismatch occurrences of P in T" are the starting positions of substrings of T that are within Hamming distance k of P, and $\mathsf{hd}(P, Q^\infty)$ denotes the Hamming distance between P and the prefix of Q^∞ of length $|P|$.

The approximate pattern matching algorithms of Charalampopoulos et al. [89] also distinguish between the easy aperiodic case and the approximately periodic case. In Chapter 9, we use a similar approach to design a small-space algorithm for pattern matching with k mismatches in the read-only setting.

In Chapter 5, we study exact and approximate pattern matching in *strings with few wildcards*. We give a result on the structure of occurrences in this setting, and use it to obtain efficient algorithms based on a similar aperiodic/periodic approach.

3.3 A Unified Approach to Pattern Matching

An important part of the work presented in this thesis (Chapters 4, 5, 6 and 9) builds on the `PILLAR` framework, introduced by Charalampopoulos, Kociumaka, and Wellnitz [89]

as a unified approach to pattern matching. In this section, we present this framework and highlight its interest.

Charalampopoulos, Kociumaka, and Wellnitz [89] observe that many efficient pattern matching algorithms can be expressed using a small set of high-level string primitives, plus some operations unrelated to strings, such as sorting numbers. They subsequently introduce the `PILLAR` model, a high-level string-manipulation language. In this model, we are given a family $\mathcal{X}$ of strings, the basic objects are *handles* (abstract pointers) to substrings $X[i..j]$ of strings $X \in \mathcal{X}$, and we can perform the following operations:

- Internal Pattern Matching $\mathsf{IPM}(P, T)$: under the condition that $|T| \leq 3|P|/2$, compute the occurrences of P in T, represented as an arithmetic progression with difference $\mathsf{per}(P)$,
- Longest Common Prefix $\mathsf{LCP}(S, T)$ (also called Longest Common Extension, LCE): compute the length of the longest common prefix of S and T,
- Longest Common Suffix $\mathsf{LCP}^R(S, T)$: compute the length of the longest common *suffix* of S and T,
- $\mathsf{Substring}(S, i, j)$: return a handle to the string $S[i..j]$,
- $\mathsf{Access}(S, i)$: return $S[i]$,
- $\mathsf{Length}(S)$: return $|S|$.

Using this set of basic operations, we can implement other common string operations with little overhead. For example, given handles to strings S, T and Q, the following operations can be implemented using $O(1)$ `PILLAR` operations ([89, Lemmas 2.5 to 2.8]):

- Test if $S = T$,
- Compute $\mathsf{per}(S)$, the period of S, or declare that S is not periodic,
- Compute $\mathsf{LCP}(S, Q^m)$, for any integer $m \geq 0$.

Using the "standard trick" of pattern matching, we can also implement pattern matching of P in T without the condition that $|T| \leq 3|P|/2$, using $O(|T|/|P|)$ `PILLAR` operations.

Landau and Vishkin [226] gave an algorithm running in time $O(n + k^2)$ for computing the edit distance between two strings of length n when the distance is at most k. By inspecting their algorithm, we can restate their result in the `PILLAR` model as follows:

Fact 3.3.1. *Let S, T be strings and k be a positive integer. There is an algorithm using $O(k^2)$ `PILLAR` operations and $O(k^2)$ space that computes the edit distance between S and T if it is less than k or reports that it is greater than k.*

When expressed in the `PILLAR` framework, algorithms are *generic* over the underlying implementation of the operations: one can choose the implementation that best fits their use case.

For example, the `PILLAR` operations can be implemented very efficiently in the RAM model: the suffix tree (Section 2.4) uses $O(n)$ space, and after $O(n)$ preprocessing, it can be used to answer any `PILLAR` operation in constant time. (Here, n is the sum of the lengths of the strings in $\mathcal{X}$.) As a consequence, an algorithm $\mathcal{A}$ using space s, p pillar operations plus t additional time can be implemented using $O(n + t + p)$ time and $O(n + s)$ space in the RAM model. For example, the algorithm of Fact 3.3.1 runs in time $O(n + k^2)$ in this model: this is the original implementation of Landau and Vishkin [226]. However, this is not the only possible implementation of `PILLAR` operations in the RAM model: these operations can also be implemented in constant extra space and linear time (without any preprocessing) in the read-only setting. The algorithm $\mathcal{A}$ would then run in $O(np + t)$ time and $O(s)$ extra space.

Furthermore, the `PILLAR` operations can also be implemented in other frameworks. An interesting example is that of compressed strings: we assume that the strings in $\mathcal{X}$ are in a

grammar-compressed form: the sum of the lengths of the strings is n but the total size of the grammars is $g \ll n$ (for highly compressible strings, g can be as small as $O(\log n)$). In this setting, there is a probabilistic construction such that, after a preprocessing phase that takes $O(g \log n)$ time, we can perform the PILLAR operations in time $O(\log n)$ [89, 125]: the above algorithm $\mathcal{A}$ would therefore run in time $O((g+p)\log n+t)$ given compressed strings as input, with an error probability of at most $1/n^c$ for some constant c. Similarly, the algorithm of Fact 3.3.1 would take $O((g+k^2)\log n)$ time. This result is not immediately visible from the result of Landau and Vishkin [226], which uses a specific RAM data structure to implement the PILLAR operations; whereas stating algorithms in the PILLAR model unlocks the use of other implementations.

Other frameworks of interest for the PILLAR model include the dynamic setting [160], which additionally allows concatenation or reversal of strings, and the quantum setting.

As a result, stating an algorithm in the PILLAR framework yields an efficient algorithm not only in the RAM model, but also in all other settings. Furthermore, this point of view opens up another line of research, that of finding efficient implementations of PILLAR operations in other models of interest. Two of the three chapters in this part follow this line of research. In Chapter 4, we give an implementation of the IPM operation for small-space algorithms, the missing piece for a full implementation of the PILLAR model in this setting. In Chapter 6, we design a data structure for LCP and LCP^R in strings with *wildcards*, thereby enabling fast pattern matching algorithms in such strings.

3.4 Organization of this Part

3.4.1 Time-space trade-off for internal pattern matching

In Chapter 4, we give an efficient data structure with tunable memory footprint for a generalization of the IPM operation in the read-only setting. More specifically, given random access to a string S of length n, we show that for any $\tau = O(n/\log^2 n)$, there exists a data structure using $\tilde{O}(n/\tau)$ space that can answer IPM queries in S in time $O(\tau)$ after $\tilde{O}(n)$-time preprocessing. Our construction is based on a 3D range-emptiness data structure built on top of a sparse suffix tree of the input string. The number of suffixes in this tree depends on the parameter τ; we give an efficient algorithm to select these suffixes using the τ-partitioning sets of Kosolobov and Sivukhin [215].

This contribution is significant because it is the missing piece for a complete implementation of the PILLAR operations in small space. The Substring, Access and Length operations can be trivially implemented in constant time and space, and Kosolobov and Sivukhin [215] gave a data structure with an $\langle O(\tau), O(n/\tau)\rangle$ time-space trade-off and quasilinear preprocessing for LCP (and LCP^R) queries. Our result gives a similar trade-off for IPM queries, showing that for any τ, there is an implementation of the PILLAR model with $O(\tau)$ time per operation using $\tilde{O}(n/\tau)$ space.

We also use our IPM data structure to design space-efficient algorithms for the Longest Common Substring (LCS) and Circular Pattern Matching problems in small-space settings (streaming, asymmetric streaming, and read-only models).

The results presented in this chapter appeared in an article published at CPM'24 [51], co-authored with P. Charalampopoulos and T. Starikovskaya. This paper won the Best Paper Award of the conference.

3.4.2 Pattern Matching with mismatches and wildcards

In Chapter 5, we study the problem of approximate pattern matching under the Hamming distance (also known as pattern matching with mismatches) in strings that may contain wildcards.

We consider the case where the pattern P contains D wildcards distributed in G contiguous groups, and work under the assumption that $n \leq 3m/2$. Our contribution is twofold. First, we characterize the structure of occurrences in this setting by showing that they can be partitioned into $O((D+k)\cdot(G+k))$ arithmetic progressions. We give a comparable lower bound by constructing an infinite family of examples that require $\Omega((D+k)k)$ arithmetic progressions to cover all approximate occurrences, building on a combinatorial result on progression-free sets by Elkin [127]. Second, we give an efficient algorithm for pattern matching with mismatches and wildcards. Our algorithm uses $O((D+k)(G+k)\cdot n/m)$ `PILLAR` operations. By exploiting the different implementations of the `PILLAR` model, this result implies efficient algorithms in several settings, including an $O(n+(D+k)(G+k)\cdot n/m)$-time algorithm in the RAM model. Our characterization and algorithm are based on a sliding-window approach that exploits the structure of groups of wildcards.

The results presented in this chapter appeared in an article published at ESA'24 [52], co-authored with P. Charalampopoulos and T. Starikovskaya.

3.4.3 Longest Common Extension with Wildcards

In Chapter 6, we give a data structure for LCE queries on strings with wildcards, called LCEW. As in Chapter 4, our result provides a time-space trade-off: for strings with G groups of wildcards and for any $t \leq G$, we show that there exists a data structure using $O(nG/t)$ space that answers LCEW queries in $O(t)$ time. Our result is based on an efficient dynamic programming scheme using the algorithm of Clifford and Clifford [100] for pattern matching with wildcards, which allows us to create a data structure that "interpolates" between the data structure of Crochemore et al. [117] and the "kangaroo jumping" technique of Landau and Vishkin [224].

We then show that this data structure enables efficient algorithms for approximate pattern matching and analysis of strings with wildcards. First, we obtain an algorithm for approximate pattern matching under the edit distance in strings with wildcards that runs in $\tilde{O}(n\sqrt{Gk})$ time, improving over a previous result by Akutsu [20]. Second, we show algorithms that run in $\tilde{O}(n\sqrt{G})$-time for computing variants of the prefix and border arrays in strings with wildcards, improving over the previous $\tilde{O}(n\sqrt{n})$-time algorithm of Iliopoulos and Radoszewski [180]. Furthermore, we unveil a surprising connection between LCE queries in strings with wildcards and (sparse) Boolean matrix multiplication. As a result, we obtain a lower bound on the preprocessing-query time product of any data structure for LCEW, and a deterministic combinatorial algorithm for sparse Boolean matrix multiplication.

The results presented in this chapter appeared in an article published at ESA'24 [50], co-authored with P. Charalampopoulos and T. Starikovskaya.

Chapter 4

Time-space trade-off for Internal Pattern Matching

4.1 Introduction

In the text indexing problem, the task is to preprocess a text T into a data structure (index) that can answer the following queries efficiently: Given a pattern P, find the occurrences of P in T. The Internal Pattern Matching problem (IPM) is a variant of the text indexing problem, where both the pattern P and the text T are fragments of a longer string S, given in advance.

Introduced in 2009 [193], IPM queries are a cornerstone of the family of internal queries on strings. The list of internal queries, primarily executed through IPM queries, comprises of period queries, prefix-suffix queries, periodic extension queries, and cyclic equivalence queries; see [202, 203, 205]. Other problems that have been studied in the internal setting include shortest unique substring [13], longest common substring [36], suffix rank and selection [40, 202], BWT substring compression [40], shortest absent string [44], dictionary matching [87, 90, 119], string covers [118], masked prefix sums [121], circular pattern matching [183], and longest palindrome [240].

The primary distinction between the classical and internal string queries lies in how the pattern is handled during queries. In classical queries, the input is explicitly provided at query time, whereas in internal queries, the input is specified in constant space using their endpoints as fragments of S. This distinction enables notably faster query times in the latter setting, as there is no need to read the input when processing the query. This characteristic of IPM and similar internal string queries renders them particularly valuable for bulk processing of textual data. This is especially advantageous when S serves as input for another algorithm, as illustrated by multiple direct and indirect (via other internal queries) applications of IPM: pattern matching with variables [132, 217], detection of gapped repeats and subrepetitions [159, 212], approximate period recovery [34, 37], computing the longest unbordered substring [206], dynamic repetition detection [35], computing string covers [118], identifying two-dimensional maximal repetitions, enumeration of distinct substrings [88], dynamic longest common substring [36], approximate pattern matching [89, 94], approximate circular pattern matching [92, 93], (approximate) pattern matching with wildcards [52], RNA folding [120], and the language edit distance problem for palindromes and squares [49].

Below we assume $|T| < 3|P|/2$, which guarantees that the set of occurrences of P in T forms an arithmetic progression and can be thus represented in $O(1)$ space.

With no preprocessing ($O(1)$ extra space), IPM queries on a string S of length n can

be answered in $O(n)$ time by a constant-space pattern matching algorithm (see [78] and references therein). On the other side of the spectrum, Kociumaka et al. [203] showed that for every string $S \in [0..\sigma]^n$, there exists a data structure of size $O(n/\log_\sigma n)$ which answers IPM queries in optimal $O(1)$ time and can be constructed in $O(n/\log_\sigma n)$ time given the packed representation of S (meaning that S divided into blocks of $\log_\sigma n$ consecutive letters, and every block is stored in one machine word). The problem has been equally studied in the compressed and dynamic settings [89, 195, 196].

4.1.1 Our Main Contribution: Small-space IPM

As our main contribution, we provide a trade-off between the constant-space and $O(n)$ query time and the $O(n/\log_\sigma n)$-space and constant query time data structures. We consider the IPM problem in the read-only setting, where one assumes random read-only access to the input string(s) and only accounts for the extra space, that is, the space used by the algorithm/data structure on top of the space needed to store the input.

Corollary 4.1.1. *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built in $O(n\log_{n/\tau} n + (n/\tau)\cdot\log^4 n\log\log n)$ time using $O((n/\tau)\cdot\log n(\log\log n)^3)$ extra space, and can answer the following internal pattern matching queries in time $O(\tau + \log n\log^3\log n)$: given $p, p', t, t' \in [1..n]$ such that $t'-t \le 2(p'-p)$, return all occurrences of $P = S[p..p']$ in $T = S[t..t']$.*

Our data structure is nearly optimal: First, when n/τ is polynomial, the construction time is linear; and secondly, the product of the query time and space of our data structure is optimal up to polylogarithmic factors (Lemma 4.3.11).

Technical overview for IPM queries. Our solution relies heavily on utilizing the concept of τ-partitioning sets, as introduced by Kosolobov and Sivukhin [215]. For a string of length n, a τ-partitioning is a subset of $O(n/\tau)$ positions that satisfies some density and consistency criteria. We use the positions of such a set as anchor points for identifying pattern occurrences, provided that the pattern avoids a specific periodic structure. To detect these anchored occurrences, we employ sparse suffix trees alongside a three-dimensional range searching structure. In cases where the pattern does not avoid said periodic structure, we employ a different strategy, leveraging the periodic structure to construct the necessary anchor points.

We next provide a brief comparison of the outlined approach with previous work. String anchoring techniques have been proven very useful in and been developed for text indexing problems, such as the longest common extension (LCE) problem, in small space [69, 215]. One of the most technically similar works to ours is that of Ben-Nun et al. [59] who considered the problem of computing a long common substring of two input strings in small space. They use an earlier variant of τ-partitioning sets, due to Birenzwige et al. [69], that has slightly worse guarantees than that of Kosolobov and Sivukhin [215]. The construction of anchors for substrings with periodic structure is quite similar to that of Ben-Nun et al. [59]. After computing a set of anchors, they aim to identify a synchronised pair of anchors that yields a long common substring; they achieve this via mergeable AVL trees. As IPM queries need to be answered in an online manner, we instead construct an appropriate orthogonal range searching data structure over a set of points that correspond to anchors. Using orthogonal range searching is a by-now classical approach for text indexing, see [229] for a survey.

4.1.2 Applications

Several internal queries reduce to IPM queries, and hence we obtain efficient implementations of them in the small-space setting. Additionally, we port several efficient approximate pattern matching algorithms to the small-space setting since IPM was the only primitive operation that they rely on that did not have an efficient small-space implementation to this day. See Section 4.4 for details on these applications.

Longest Common Substring (LCS). The LCS problem is formally defined as follows.

Problem 4.1.2 (Longest Common Substring (LCS)).
▷ **Input:** Strings S and T of length at most n.
▷ **Output:** The length of a longest string that appears as a (contiguous) fragment in both S and T.

The length of the longest common substring is one of the most popular string-similarity measures. The by-now classical approach to the LCS problem is to construct the suffix tree of S and T in $O(n)$ time and space. The longest common substring of the two strings appears as a common prefix of a pair of suffixes of S and T and hence its length is the maximal string-depth of a node of the suffix tree with leaf-descendants corresponding to suffixes of both strings; this node can be found in $O(n)$ time in a bottom-up manner.

Starikovskaya and Vildhøj [273] were the first to consider the problem in the read-only setting. They showed that for any $n^{2/3} < \tau \leq n$, the problem can be solved in $O(\tau)$ extra space and $O(n^2/\tau)$ time. Kociumaka et al. [204] extended their bound to all $1 \leq \tau \leq n$, which in particular resulted in a constant-space read-only algorithm running in time $\tilde{O}(n^2)$.

In an attempt to develop even more space-efficient algorithms for the LCS problem, it might be tempting to consider the streaming setting, which is particularly restrictive: in this setting, one assumes that the input arrives letter-by-letter, as a stream, and must account for all the space used. Unfortunately, this setting does not allow for better space complexity: any streaming algorithm for LCS, even randomised, requires $\Omega(n)$ bits of space (Theorem 4.5.3). In the asymmetric streaming setting, which is slightly less restrictive and was introduced by Andoni et al. [38] and Saks and Seshadhri [266], the algorithm has random access to one string and sequential access to the other. Mai et al. [233] showed that in this setting, LCS can be solved in $\tilde{O}(n^2)$ time and $O(1)$ space. By utilising (a slightly more general variant of) IPM queries, we extend their result and show that for every $\tau \in [\sqrt{n}\log n(\log\log n)^3..n]$, there is an asymmetric streaming algorithm that solves the LCS problem in $O(\tau)$ space and $\tilde{O}(n^2/\tau)$ time (Theorem 4.6.1). Note that these bounds almost match the bounds of Kociumaka et al. [204], while the setting is stronger.

Circular Pattern Matching (CPM). The CPM problem is formally defined as follows.

Problem 4.1.3 (Circular Pattern Matching (CPM)).
▷ **Input:** A pattern P of length m, a text T of length n.
▷ **Output:** All occurrences of rotations of P in T.

The interest in occurrences of rotations of a given pattern is motivated by applications in Bioinformatics and Image Processing: in Bioinformatics, the starting position

of a biological sequence can vary significantly due to the arbitrary nature of sequencing in circular molecular structures or inconsistencies arising from different standards of linearization applied to sequence databases; and in Image Processing, the contour of a shape can be represented using a directional chain code, which can be viewed as a circular sequence, particularly when the orientation of the image is irrelevant [39].

For strings over an alphabet of size σ, the classical read-only solution for CPM via the suffix automaton of $P \cdot P$ runs in $O(n \log \sigma)$ time and uses $O(m)$ extra space [230]. Recently, Charalampopoulos et al. showed a simple $O(n)$ time and $O(m)$ extra space solution. The problem has been also studied from the practical point of view [97, 143, 275] and in the text indexing setting [176, 177, 182].

It is not hard to see that the CPM and the LCS problems are closely related: occurrences of rotations of P in T are exactly the common substrings of $P \cdot P$ and T of length m. Implicitly using this observation, we show that in the streaming setting, the CPM problem requires $\Omega(m)$ bits of space (Theorem 4.5.4) and that in the asymmetric streaming setting, for every $\tau \in [\sqrt{m} \log m (\log \log m)^3 . . m]$, there exists an algorithm that solves the CPM problem in time $\tilde{O}(mn/\tau)$ using $O(\tau)$ extra space (Corollary 4.6.5). Finally, in the read-only setting, we give an *online* $O(n)$-time, $O(1)$-space algorithm (Theorem 4.7.1).

4.2 Preliminaries

We use the definitions and notations related to strings that are defined in Section 2.1 and the basic data structures for string processing presentend in Section 2.4. Recall that a suffix tree for a string S is essentially a compact trie representing the set of all suffixes of S, whereas a sparse suffix tree contains only a subset of these suffixes (see e.g. [230]). We restate here a few other results on efficient string algorithms that we will later use.

Fact 4.2.1 ([215, Theorem 3]). *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $b = \Omega(\log^2 n)$, one can construct in $O(n \log_b n)$ time and $O(b)$ space the sparse suffix tree for b arbitrarily chosen suffixes of S.*

Fact 4.2.2 ([78]). *There is a read-only online algorithm that finds all occurrences of a pattern P of length m in a text T of length $n \geq m$ in $O(n)$ time and $O(1)$ space.*

Fact 4.2.3 ([139, Lemma 6]). *Given read-only random access to a string S of length n, one can decide in $O(n)$ time and $O(1)$ space if S is periodic and, if so, compute $\mathsf{per}(S)$.*

Fact 4.2.4 ([124]). *Given read-only random access to a string S of length n, the lexicographically smallest rotation of a string S can be computed in $O(n)$ time and $O(1)$ space.*

Static predecessor.

For a static set, a combination of x-fast tries [285] and deterministic dictionaries [259] yields the following efficient deterministic data structure; cf. [137].

Fact 4.2.5 ([137, Proposition 2]). *A sorted static set $Y \subseteq [1..U]$ can be preprocessed in $O(|Y|)$ time and space so that predecessor queries can be performed in $O(\log \log |U|)$ time.*

Weighted ancestor queries.

Let $\mathcal{T}$ be a rooted tree with integer weights on nodes. A *weighted ancestor* query for a node u and weight d must return the highest ancestor of u with weight at least d.

Fact 4.2.6 ([33]). *Let $\mathcal{T}$ be a rooted tree of size n with integer weights on nodes. Assume that each weight is at most n, with the weight of the root being zero, and the weight of every non-root node being strictly larger than its parent's weight. $\mathcal{T}$ can be preprocessed in $O(n)$ time and space so that weighted ancestor queries on it can be performed in $O(\log\log n)$ time.*

If $\mathcal{T}$ is the suffix tree of a string and the weights are the string-depths of the nodes, this result can be improved further:

Fact 4.2.7 ([58]). *The suffix tree $\mathcal{T}$ of a string of length n can be preprocessed in $O(n)$ time and $O(n)$ space so that weighted ancestor queries on it can be performed in $O(1)$ time.*

3D range emptiness.

For a positive integer U, let $[U]$ be a shorthand for $[1..U]$, and consider a set $S \subseteq [U]^3$. A three-dimensional orthogonal range emptiness query asks whether a range $[a_1..a_2] \times [b_1..b_2] \times [c_1..c_2]$ intersects S.

Fact 4.2.8 ([191, Theorem 2]). *Let S be an n-points subset of $[U]\times[U]\times[U]$. There exists a data structure that answers three-dimensional orthogonal range emptiness queries on S in $O(\log\log U+(\log\log n)^3)$ time, uses $O(n\log n(\log\log n)^3)$ space, and can be constructed in $O(n\log^4 n\log\log n)$ time. If the query range intersects S, the data structure also outputs a witness point contained in the intersection.*

Remark 4.2.9. Better space vs. query-time trade-offs than the above are known for the 3D range emptiness problem; cf [84] and references therein. We opted for the data structure encapsulated by Fact 4.2.8 due to its efficient construction algorithm. Note that a data structure capable of reporting all points in an orthogonal range over a $[U] \times [U] \times [U]$ grid with n points in time $O(Q_1(U,n)+Q_2(U,n)\cdot|\mathsf{output}|)$ can answer range emptiness queries, also returning a witness in the case the range is not empty, in time $O(Q_1(U,n)+Q_2(U,n))$.

4.3 Internal Pattern Matching

We consider a slightly more powerful variant of IPM queries, as required by our applications. A reader that is only interested in IPM queries can focus on the case when $a = \varepsilon$.

Problem 4.3.1 (Extended IPM).
▷ **Input:** A string S of length n over an integer alphabet to which we have read-only random access.
▷ **Query:** Given $p, p', t, t' \in [1..n]$ and $a \in \Sigma \cup \{\varepsilon\}$, return whether $P := S[p..p']a$ occurs in $T := S[t..t']$ and, if so, return a witness occurrence.

Our solution for Extended IPM heavily relies on a solution for the following auxiliary problem.

Problem 4.3.2 (Anchored IPM)**.**
▷ **Input:** A string S of length n (over an integer alphabet Σ) to which we have read-only random access and a set $\mathcal{A} \subseteq [1..n]$.
▷ **Query:** Given $p, x, p', t, t' \in [1..n]$ with $p \leq x \leq p'$, $x \in \mathcal{A}$, and $a \in \Sigma \cup \{\varepsilon\}$, for $P := S[p..p']a$, decide whether there exists an occurrence of P at some position $j \in [t..t' - |P| + 1]$ such that $j + (x - p) \in \mathcal{A}$ and, if so, return a witness.

The anchored index $x \in \mathcal{A}$ appears at position $x - p + 1$ in $S[p..p']$. The constraint $j + (x - p) \in \mathcal{A}$ means that the occurrence $S[j..j + |P| - 1]$ of P that we are searching for must also contain an anchored index in its $x - p + 1$-th position, which is position $j + (x - p)$ in S.

Lemma 4.3.3. *There exists a data structure for the* Anchored IPM *problem that can be built using* $O(n \log_{|\mathcal{A}|} n) + O(|\mathcal{A}| \log^4 |\mathcal{A}| \log\log |\mathcal{A}|)$ *time and* $O(|\mathcal{A}| \log |\mathcal{A}| (\log\log |\mathcal{A}|)^3)$ *extra space, and answers queries in* $O(\log^3 \log n)$ *time.*

Proof. For an integer $y \in [1..n]$, denote $P_y := \mathrm{rev}(S[..y))$ and $S_y := S[y..]$. Consider the family $\mathcal{X} := \{(P_y\$, S_y\$) : y \in \mathcal{A}\}$ of pairs of strings, where $\$ \notin \Sigma$ is a letter lexicographically smaller than all others. Using Fact 4.2.1, we build a sparse suffix tree RSST for the first components of the elements of $\mathcal{X}$ and a sparse suffix tree SST for the second components of the elements of $\mathcal{X}$.

Consider a three-dimensional grid $[1..n] \times [1..n] \times [1..n]$. In this grid, create a set Π of points, which contains, for each element $(P_y\$, S_y\$)$ of $\mathcal{X}$, a point $(\mathsf{rank}_{\mathrm{rev}}(y), \mathsf{rank}(y), y)$, where $\mathsf{rank}_{\mathrm{rev}}(y)$ is the lexicographic rank of $P_y\$$ among the first components of the elements of $\mathcal{X}$ and $\mathsf{rank}(y)$ is the lexicographic rank of $S_y\$$ among the second components of the elements of $\mathcal{X}$.

Given a query (p, x, p', t, t', a), we first retrieve the leaves corresponding to $P_x\$$ and $S_x\$$ in RSST and SST, respectively. This can be done in $O(\log\log n)$ time with the aid of Fact 4.2.5 built over the elements of $\mathcal{A}$, with $x \in \mathcal{A}$ storing pointers to the corresponding leaves as satellite information. Next, we retrieve the (possibly implicit) nodes u and v corresponding to $\mathrm{rev}(S[p..x))$ in RSST and $S[x..p']a$ in SST, respectively. This can be done in $O(\log\log n)$ time after an $O(|\mathcal{A}|)$-time preprocessing of (a) the two trees according to Fact 4.2.6 and (b) the edge-labels of the outgoing edges of each node using Fact 4.2.5. Now, it suffices to check if there is some integer j such that the leaf corresponding to $P_j\$$ is a descendant of u, the leaf corresponding to $S_j\$$ is a descendant of v, and $j \in [t + (x - p)..t' - (p' + |a| - x)]$. After a linear-time bottom-up preprocessing of RSST and SST, we can retrieve in $O(1)$ time the following ranges:

- $R_1 = \{\mathsf{rank}_{\mathrm{rev}}(y) : \text{the node of RSST corresponding to } P_y\$ \text{ is a descendant of } u\}$;
- $R_2 = \{\mathsf{rank}(y) : \text{the node of SST corresponding to } S_y\$ \text{ is a descendant of } v\}$.

The query then reduces to deciding whether the orthogonal range $R_1 \times R_2 \times [t + (x - p)..t' - (p' + |a| - x)]$ contains any point in Π, and returning a witness if it does. We can do this efficiently by building the data structure encapsulated in Fact 4.2.8 for Π: the query time is $O(\log^3 \log n)$, while the construction time is $O(n \log_{|\mathcal{A}|} n) + O(|\mathcal{A}| \log^4 |\mathcal{A}| \log\log |\mathcal{A}|)$ and the space usage is $O(|\mathcal{A}| \log |\mathcal{A}| (\log\log |\mathcal{A}|)^3)$. □

For an integer parameter τ, we next present a data structure for Extended IPM that uses $\tilde{O}(n/\tau)$ space on top of the space required to store S and answers queries in nearly-constant time provided that P is of length greater than 5τ. We achieve this result using the so-called τ-partitioning sets of Kosolobov and Sivukhin [215] as *anchors* for the occurrences if P avoids a certain periodic structure, and by exploiting said periodic structure to construct anchors in the remaining case.

Definition 4.3.4 (τ-partitioning set)**.** *Given an integer $\tau \in [4..n/2]$, a set of positions $\mathcal{P} \subseteq [1..n]$ is called a* τ-partitioning set *if it satisfies the following properties:*

a) if $S[i-\tau..i+\tau] = S[j-\tau..j+\tau]$ for $i, j \in [\tau+1..n-\tau]$, then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$;

b) if $S[i..i+\ell] = S[j..j+\ell]$, for $i, j \in \mathcal{P}$ and some $\ell \geq 0$, then, for each $d \in [0..\ell-\tau)$, $i+d \in \mathcal{P}$ if and only if $j+d \in \mathcal{P}$;

c) if $i, j \in [1..n]$ with $j-i > \tau$ and $(i..j) \cap \mathcal{P} = \emptyset$, then $S[i..j]$ has period at most $\tau/4$.

Fact 4.3.5 ([215])**.** *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4..O(n/\log^2 n)]$ and $b = n/\tau$, one can construct in $O(n \log_b n)$ time and $O(b)$ extra space a τ-partitioning set $\mathcal{P}$ of size $O(b)$. The set $\mathcal{P}$ additionally satisfies the property that if a fragment $S[i..j]$ has period at most $\tau/4$, then $\mathcal{P} \cap [i+\tau..j-\tau] = \emptyset$.*

Definition 4.3.6 (τ-runs)**.** *A fragment F of a string S is a τ-run if and only if $|F| > 3\tau$, $\mathsf{per}(F) \leq \tau/4$, and F cannot be extended in either direction without its period changing. The Lyndon root of a τ-run R is the lexicographically smallest rotation of $R[1..\mathsf{per}(R)]$.*

The following fact follows from the proof of Lemma 10 in the full version of [91], where the definition of τ-runs is slightly different, but captures all of our τ-runs.

Fact 4.3.7 (cf. [91, proof of Lemma 10])**.** *Two τ-runs can overlap by at most $\tau/2$ positions. The number of τ-runs in a string of length n is $O(n/\tau)$.*

Lemma 4.3.8. *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau \in [4..O(n/\log^2 n)]$, all τ-runs in S can be computed and grouped by Lyndon root in $O(n \log_b n)$ time using $O(b)$ extra space, where $b = n/\tau$. Within the same complexities, we can compute, for each τ-run, the first occurrence of its Lyndon root in it.*

Proof. We first compute a τ-partitioning set $\mathcal{P}$ for S using Fact 4.3.5. Due to Property c), its converse that is stated in Fact 4.3.5, and Fact 4.3.7 there is a natural injection from the τ-runs to the maximal fragments of length at least τ that do not contain any position in $\mathcal{P}$ — the τ-run corresponding to such a maximal fragment may extend for τ more positions in each direction. We can find the period of each maximal fragment in time proportional to its length using $O(1)$ extra space due to Fact 4.2.3. We then try to extend the maximal fragment to a τ-run using $O(\tau)$ letter comparisons. Additionally, we compute the Lyndon root of each computed τ-run R in $O(\tau) = O(|R|)$ time by applying Fact 4.2.4 to $R[1..\mathsf{per}(R)]$. The first occurrence of the Lyndon root in the τ-run can be computed in constant time since we know which rotation of $R[1..\mathsf{per}(R)]$ equals the Lyndon root. Over all τ-runs, the total time is $O(n)$ due to Fact 4.3.7. □

We next prove the main result of this section.

Theorem 4.3.9. *For any $\ell \in [20..O(n/\log^2 n)]$, there is a data structure for* Extended IPM *that can be built using $O(n \log_{n/\ell} n) + O((n/\ell) \cdot \log^4 n \log\log n)$ time and $O((n/\ell) \cdot \log n (\log\log n)^3)$ extra space given random access to S and answers queries in $O(\log^3 \log n)$ time, provided that $|P| > \ell$.*

Proof. Let $\tau = \lfloor \ell/5 \rfloor$. We use Fact 4.3.5 and Lemma 4.3.8 with parameter τ to compute a partitioning set $\mathcal{P}$ of size $O(n/\tau)$ and all τ-runs in S, grouped by Lyndon root, each

one together with the first occurrence of its Lyndon root. We create a static predecessor structure $\mathcal{R}$ using Fact 4.2.5, where we insert the starting position of each run R with the following satellite information: R's ending position, the first occurrence of R's Lyndon root in R, and an identifier of its group. We additionally create a data structure $\mathcal{Q}$, where, for each group of τ-runs with a common root L, indexed by their identifiers, we construct, using Fact 4.2.5, a predecessor data structure for a set $Q_L := \{(y, s, e) : S[s..e]$ is the longest τ-run with a suffix $L \cdot L[1..y]\}$, with the first components being the keys and the remaining components being stored as satellite information. The sets Q_L can be straightforwardly constructed in $O(n \log n/\tau)$ time.

Now, let $\mathcal{L}$ be a set that contains the ending position of each τ-run as well as the starting (resp. ending) positions of the first (resp. last) two occurrences of the Lyndon root in this τ-run; $\mathcal{L}$ can be straightforwardly constructed in $O(n/\tau)$ time given the information returned by the application of Lemma 4.3.8. We then construct a set $\mathcal{A} := \mathcal{P} \cup \mathcal{L}$ and preprocess the string S and the set $\mathcal{A}$ according to Lemma 4.3.3.

Our query comprises of two steps.

Step 1: First, we deal with the case when both P and T have period at most $\tau/4$. Since P and T are of length at least 5τ, due to Fact 4.3.7, each of them can be only contained in the τ-run whose starting position is closest to it in the left. We can thus check whether they both have period at most $\tau/4$ in $O(\log \log n)$ time by performing two predecessor queries on $\mathcal{R}$. If this turns out to be the case, we then check whether the two corresponding τ-runs belong to the same group. If they do not, then P does not occur in T due to Fact 4.3.7. Otherwise, let the common Lyndon root of the two runs be L. We can compute in constant time non-negative integers $x_P, x_T, y_P, y_T < |L|$ and e_P, e_T such that $P = L(|L| - x_P..] \cdot L^{e_P} \cdot L[..y_P]$ and $T = L(|L| - x_T..] \cdot L^{e_T} \cdot L[..y_T]$. Note that P occurs in T if and only if at least one of the following conditions is met: (1) $e_P = e_T$, $x_P \leq x_T$, and $y_P \leq y_T$; or (2) $e_P = e_T - 1$ and $x_P \leq x_T$; or (3) $e_P = e_T - 1$ and $y_P \leq y_T$; or (4) $e_P \leq e_T - 2$. In each of the four cases, we can compute an occurrence of P in T in constant time.

Step 2: In the second step of the query, we consider the case when $\mathsf{per}(T) > \tau/4$ and distinguish between two cases depending on whether $\mathsf{per}(S[p..p + 3\tau]) \leq \tau/4$. In each case, it suffices to perform at most two anchored internal pattern matching queries.

Case I: $\mathsf{per}(S[p..p + 3\tau]) > \tau/4$. Due to Property c), $[p..p + 3\tau] \cap \mathcal{P} \neq \emptyset$. Let $x = \min([p..p + 3\tau] \cap \mathcal{P})$. Additionally, due to Property b), for any occurrence of P in S at position j, we have $[p..p+3\tau] \cap \mathcal{P} = (p-j) + ([j..j + 3\tau] \cap \mathcal{P})$, and hence $j + (x - p) \in \mathcal{P}$. Thus, an anchored IPM query returns the desired answer in $O(\log^3 \log n)$ time.

Case II: $\mathsf{per}(S[p..p + 3\tau]) \leq \tau/4$. We distinguish between two subcases depending on whether $\mathsf{per}(P) > \tau/4$; we can check this in $O(\log \log n)$ time with the aid of data structure $\mathcal{R}$ by comparing p' with the ending position of the τ-run that contains $S[p..p + 3\tau]$ and checking if $a = P[|P| - \mathsf{per}(S[p..p + 3\tau])]$ if $a \neq \varepsilon$.

Subcase (a): $\mathsf{per}(P) > \tau/4$. In this case, for any occurrence of P in T, the ending position of the τ-run that is a prefix of P must be aligned with the ending position of a τ-run in T, which belongs to $\mathcal{L} \subseteq \mathcal{A}$.

Recall that $P = S[p..p']a$. If the period of $S[p..p']$ is greater than $\tau/4$, we retrieve the ending position of the τ-run containing $S[p..p + 3\tau]$, which is in $\mathcal{L} \subseteq \mathcal{A}$ as well and issue

an anchored internal pattern matching query. Assume now that the period of $S[p..p'+1]$ is at most $\tau/4$ and $\varepsilon \neq a \neq P[|P| - \mathsf{per}(S[p..p'])]$, in which case p' might not be in $\mathcal{A}$. In this case, we retrieve a fragment $S[q..q']$ equal to $S[p..p']$, such that q' is an ending position of a τ-run in $O(\log\log n)$ time using the data structure $\mathcal{Q}$, if such a fragment exists, and use $q \in \mathcal{L} \subseteq \mathcal{A}$ as the anchor to our internal anchor query, effectively searching for $S[q..q']a = P$. Observe that if such a fragment $S[q..q']$ does not exist, P cannot have any occurrence in T.

Subcase (b): $\mathsf{per}(P) \leq \tau/4$. We consider an occurrence of P in the τ-run that contains P that starts in its first $\mathsf{per}(P)$ positions and one that ends in its last $\mathsf{per}(P)$ positions. Let these two occurrences be at positions p_1 and p_2, respectively. Each of these occurrences contains at least one element of $\mathcal{L}$; let those elements be denoted q_1 for the occurrence at p_1 and q_2 for the occurrence at p_2.

Note that these elements can be straightforwardly computed given the endpoints of the τ-run, the endpoints of P, and the first occurrence of the Lyndon root in the τ-run, which we already have in hand. We then issue anchored internal pattern matching queries for $(p_1, q_1, p_1 + |P| - 1, t, t')$ and $(p_2, q_2, p_2 + |P| - 1, t, t')$ as both q_1 and q_2 are in $\mathcal{L}$. These queries are answered in $O(\log^3 \log n)$ time. As we show next, if P has an occurrence in T, this occurrence will be returned by those queries.

Consider an occurrence of P in $S[t..t']$ and denote the τ-run that contains this occurrence by R. Since $\mathsf{per}(T) > \tau/4$, R does not contain $S[t..t']$. Without loss of generality, let us assume that R does not extend to the left of $S[t..t']$, the remaining case is symmetric. Let the first occurrence of the Lyndon root L of the τ-run in P be at position $i = q_1 - p_1 + 1$ of P, noting that $i \leq \mathsf{per}(P)$. Then, in the leftmost occurrence of P in R, position i must be aligned with either the first or the second position where L occurs in R. By the construction of the set $\mathcal{L}$, it follows that both of these positions are in $\mathcal{L}$, and hence the anchored internal pattern matching query will return an occurrence. □

Corollary 4.1.1. *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built in $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time using $O((n/\tau) \cdot \log n (\log\log n)^3)$ extra space, and can answer the following internal pattern matching queries in time $O(\tau + \log n \log^3 \log n)$: given $p, p', t, t' \in [1..n]$ such that $t' - t \leq 2(p' - p)$, return all occurrences of $P = S[p..p']$ in $T = S[t..t']$.*

Proof. If the length of P is at most $\max\{\tau, 20\}$, we compute its occurrences in T, whose length is $O(\tau)$, in $O(\tau)$ time using Fact 4.2.2. In what follows, we assume that $|P| > \max\{\tau, 20\}$.

We build the Extended IPM data structure of Theorem 4.3.9 for S with $\ell = \max\{\tau, 20\}$. This allows us to efficiently answer the decision version of the desired IPM queries, also returning a witness, in $O(\log^3 \log n)$ time. If the query does not return an occurrence of P in T, we are done. Otherwise, we have to compute all occurrences of P in T represented as an arithmetic progression (cf Corollary 3.2.2). Let the witness returned by the data structure be $S[x..x']$. Consider the rightmost occurrence of P in $S[t..x')$, or, if it does not exist, the leftmost occurrence in $S(x..t']$. Such an occurrence can be found by binary search. If no such occurrence exists, we are again done, as P has a single occurrence in T. Otherwise, the occurrences of P in T form an arithmetic progression with difference equal to the difference d of x and the starting position of the found occurrence due to Corollary 3.2.2. We compute the extreme values of this arithmetic progression using binary search as well: we compute the minimum and the maximum $j \in \mathbb{Z}$ such that

$S[x + j \cdot d \,..\, x' + j \cdot d] = P$ and $t \leq x + j \cdot d \leq x' + j \cdot d \leq t'$ using $O(\log n)$ IPM queries in total; the complexity follows. □

4.3.1 Lower Bound for an IPM data structure

We now show that the product of the query time and the space achieved in Corollary 4.1.1 is optimal up to polylogarithmic factors, via a reduction from the following problem.

Problem 4.3.10 (Longest Common Extension (LCE))**.**
▷ **Input:** A string S of length n.
▷ **Query:** Given $i, j \in [1 \,..\, n]$, return the largest ℓ such that $S[i \,..\, i + \ell] = S[j \,..\, j + \ell]$.

Bille et al. [64, Lemma 4] showed that any data structure for LCE for n-length strings that uses s bits of extra space on top of the input has query time $\Omega(n/s)$.

Lemma 4.3.11. *In the non-uniform cell-probe model, any IPM data structure that uses s bits of space on top of the input for a string of length n, has query time $\Omega(n/(s \log n))$.*

Proof. We prove Lemma 4.3.11 by reducing LCE queries in a string S of length n to IPM queries in S. Consider an IPM data structure with space s and query time q and observe that IPM queries can be used to check substring equality since $S[i \,..\, i'] = S[j \,..\, j']$ if and only if $S[i \,..\, i']$ occurs inside the interval $[j \,..\, j']$ and $j' - j = i' - i$. Using binary search, we can thus answer any LCE query via $O(\log n)$ IPM queries. Hence, we have $q \log n = \Omega(n/s)$, which concludes the proof the lemma. □

Lemma 4.3.11 implies a similar lower bound for the word RAM model, which is weaker than the non-uniform cell-probe model.

4.4 Other Internal Queries and Approximate Pattern Matching

In the PILLAR model of computation [89] (presented in Section 3.3) the runtimes of algorithms are analysed with respect to the number of calls made to standard word-RAM operations and a few primitive string operations. It has been used to design algorithms for internal queries [202, 203, 205], approximate pattern matching under Hamming distance [89] and edit distance [94], circular approximate pattern matching under Hamming distance [93] and edit distance [96], and (approximate) wildcard pattern matching under Hamming distance [52]. Space-efficient implementations of the PILLAR model immediately result in space-efficient implementations of the above algorithms.

All PILLAR operations can be implemented in small space in the read-only setting. Operations other than LCE, LCE^R, and IPM admit trivial constant-time and constant-space implementations in the read-only setting. For any $\tau = O(n/\log^2 n)$, Kosolobov and Sivukhin [215] showed that after an $O(n \log_{n/\tau} n)$-time, $O(n/\tau)$-space preprocessing, LCE and LCE^R queries can be answered in $O(\tau)$ time. For IPM queries, Corollary 4.1.1 gives a similar trade-off.

In [202, 203, 205] it is (implicitly) shown that the following internal queries can be efficiently implemented in the PILLAR model.

- A *cyclic equivalence query* takes as input two equal-length fragments $U = S[i \,..\, i + \ell]$ and $V = S[j \,..\, j + \ell]$, and returns all rotations of U that are equal to V. Any cyclic equivalence query reduces to $O(1)$ LCE queries and $O(1)$ IPM(P, T) queries with $|T|/|P| = O(1)$.

- A *period query* takes as input a fragment $U = S[i..j]$, and returns all periods of U. Such a period query reduces to $O(\log |U|)$ LCE queries and $O(\log |U|)$ IPM(P, T) queries with $|T|/|P| = O(1)$.
- A *2-period* query takes as input a fragment $U = S[i..j]$, checks if U is periodic and, if so, it also returns U's period. Such a query reduces to $O(1)$ LCE queries and $O(1)$ IPM(P, T) queries with $|T|/|P| = O(1)$.

Corollary 4.4.1. *Suppose that we have read-only random access to a string S of length n over an integer alphabet. For any integer $\tau = O(n/\log^2 n)$, there is a data structure that can be built using $O(n \log_{n/\tau} n + (n/\tau) \cdot \log^4 n \log\log n)$ time and $O((n/\tau) \cdot \log n(\log\log n)^3)$ extra space and can answer cyclic equivalence and 2-period queries on S in $O(\tau + \log n \log^3 \log n)$ time, and period queries on S in $O(\tau \log n + \log^2 n \log^3 \log n)$ time.*

By plugging this implementation of the PILLAR model into [52, 89, 93, 94, 96], we obtain the following:

Corollary 4.4.2. *Suppose that we have read-only random access to a text T of length n, a pattern P of length m over an integer alphabet. Given an integer threshold k, for any integer $\tau = O(m/\log^2 m)$, we can compute:*

- *the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n + k^2\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^2)$ extra space (from [89]);*
- *the approximate occurrences of P in T under the edit distance in $\tilde{O}(n + k^{3.5}\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^{3.5})$ extra space (from [94]);*
- *the approximate occurrences of all rotations of P in T under the Hamming distance in $\tilde{O}(n + k^3\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^3)$ extra space (from [93]);*
- *the approximate occurrences of all rotations of P in T under the edit distance in $\tilde{O}(n + k^5\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + k^5)$ extra space (from [96]);*
- *in the case where P has D wildcard letters arranged in G maximal intervals, the approximate occurrences of P in T under the Hamming distance in $\tilde{O}(n + (D + k)(G + k)\tau \cdot n/m)$ time using $\tilde{O}(m/\tau + (D + k)(G + k))$ extra space (from [52]).*

To the best of our knowledge, the only work that has considered approximate pattern matching in the read-only model is due to Bathie et al. [49][1]. They presented online algorithms both for the Hamming distance and the edit distance; for the Hamming distance their algorithm uses $O(k \log m)$ extra space and $O(k \log m)$ time per letter of the text, and for the edit distance $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortised time per letter.

4.5 Lower Bounds for LCS and CPM in the Streaming Setting

In the streaming setting, we receive a stream composed of the concatenation of the input strings, e.g., the pattern and the text in the case of CPM. We account for all the space used, including the space needed to store any information about the input strings.

We exploit the well-known connection between streaming algorithms and communication complexity to prove linear-space lower bounds for streaming algorithms for LCS and CPM. Our streaming lower bounds are based on reductions from the following problem:

[1] This is the work presented in Chapter 9 of this thesis.

Problem 4.5.1 (Augmented Index)**.**
▷ **Alice** holds a binary string S of length n.
▷ **Bob** holds an index $i \in [1..n]$ and the string $S[..i-1]$.
▷ **Output:** Bob is to return the value of $S[i]$.

In the one-way communication complexity model, Alice performs an arbitrary computation on her input to create a message $\mathcal{M}$ and sends it to Bob who must compute the output using this message and his input. The communication complexity of a protocol is the size of $\mathcal{M}$ in bits. The protocol is randomized when either Alice or Bob use randomized computation.

Fact 4.5.2 ([83, Theorem 2.3])**.** *The randomized one-way communication complexity of* Augmented Index *is* $\Omega(n)$ *bits.*

Theorem 4.5.3. *In the streaming setting, any algorithm for* LCS *for strings of length at most* n *uses* $\Omega(n)$ *bits of space.*

Proof. We show the bound by a reduction from the Augmented Index problem. Consider an input $S, (i, S[..i-1])$ to the Augmented Index problem, where $|S| = n$. We observe that for $A = 0^n\$S$ and $B = 0^n\$S[..i-1]1$, where $\$ \notin \{0,1\}$, we have $\mathsf{LCS}(A,B) = n+i+1$ if and only if $S[i] = 1$. Now, if we have a streaming algorithm for LCS that uses b bits of space, we can develop a one-way protocol for the Augmented Index problem with message size b bits as follows. Alice runs the algorithm on A. When she reaches the end of A, she sends the memory state of the algorithm and n (in binary) to Bob. Bob continues running the algorithm on B, which he can construct knowing n and $S[..i-1]$, and returns 1 if and only if $\mathsf{LCS}(A,B) = n+i+1$. Fact 4.5.2 implies that $b + \log n = \Omega(n)$, and hence $b = \Omega(n)$. □

Theorem 4.5.4. *In the streaming setting, any algorithm for* CPM *uses* $\Omega(m)$ *bits of space, where* m *is the size of the pattern.*

Proof. We show the bound by a reduction from the Augmented Index problem. Consider an input $S, (i, S[..i-1])$ to the Augmented Index problem, where $|S| = m$. Let $A = S\$$ and $B = S\$S[..i-1]1$, where $\$ \notin \{0,1\}$. B ends with an occurrence of a rotation of A if and only if $S[i] = 1$. Now, if we have a streaming algorithm for CPM that uses b bits of space, we can develop a one-way protocol for the Augmented Index problem with message size b bits as follows. Alice runs the algorithm on the pattern $A = S\$$ and the first $|S|+1$ letters of the string B. She then sends the memory state of the algorithm to Bob. Bob continues running the algorithm on the remainder of B, i.e., on $S[1..i-1]1$, and returns 1 if and only if the algorithm reports an occurrence of a rotation of A ending at position $n+i+1$. By Fact 4.5.2, we have $b = \Omega(m)$. □

4.6 LCS and CPM in the Asymmetric Streaming Setting

In this section, we use Theorem 4.3.9 to show that for any $\tau \in [\tilde{\Omega}(\sqrt{m})..O(m/\log^2 m)]$, there are asymmetric streaming algorithms for LCS and CPM that use $O(\tau)$ space and $\tilde{O}(m/\tau)$ time per letter. We start by giving an algorithm for a generalization of the LCS problem that can be used to solve both LCS and CPM. For two strings S, T, a fragment $T[t..t']$ is a *T-maximal common substring* of S and T if it occurs in S and neither $T[t-1..t']$ (assuming $t > 1$) nor $T[t..t'+1]$ (assuming $t' < n$) occur in S.

Theorem 4.6.1. *Assume to be given read-only random access to a string S of length m and streaming access to a string T of length n over an integer alphabet, where $n \geq m$.*

For all $\tau \in [\sqrt{m}\log m(\log\log m)^3 . . O(m/\log^2 m)]$, there is an algorithm that reports all T-maximal common substrings of S and T using $O(\tau)$ space and $O(nm/\tau \cdot \log\log\sigma)$ time.

Proof. We cover T with windows of length 2τ (except maybe for the last) that overlap by τ letters: there are $O(n/\tau)$ such windows. After reading such a window W, we apply the procedure encapsulated in the following claim with $A = W$ and $B = S$:

▷ Claim 4.6.2. Let A, B be strings of respective lengths a and b, where $a < b < a^{O(1)}$, over an integer alphabet of size σ. Given read-only random access to A and B, we can compute all B-maximal common substrings of A and B, and the length $\mathsf{LCSuf}(A, B)$ of the longest suffix of A that occurs in B in $O(b\log\log\sigma)$ time using $O(a)$ extra space.

Claim proof. We start by building the suffix tree for A and preprocessing it for constant-time weighted ancestor queries: this takes $O(a)$ time (see Fact 4.2.1 and Fact 4.2.7). Additionally, we preprocess the labels of edges outgoing from each node according to Fact 4.2.5. Then, the algorithm traverses the tree maintaining the following invariant: at every moment, it is at a node (maybe implicit) corresponding to a substring $B[i..j]$ of B. It starts at the root of the tree with $i = 1$ and $j = 0$. In each iteration, the algorithm tries to go down the tree from the current node using $B[j+1]$; this takes $O(\log\log\sigma)$ time. If it succeeds, it increments j and continues. Otherwise, it considers two cases. If it is at the root, it increments both i and j. Otherwise, it jumps to the node corresponding to $B[i+1..j]$ via a weighted ancestor query in $O(1)$ time and increments i. The nodes reached by an edge traversal and abandoned with the use of a weighted ancestor query in the next iteration are in one-to-one correspondence with the B-maximal common substrings of A and B. The LCSuf of A and B is the depth of the deepest visited node that corresponds to a suffix of A. As at least one of the indices i, j gets incremented at every step of the traversal, the total runtime is $O(b\log\log\sigma)$. ◁

The above sliding-window procedure takes $O(m\log\log\sigma)$ time per window and uses $O(\tau)$ space, which adds up to $O(nm/\tau \cdot \log\log\sigma)$ time in total, and finds all T-maximal common substrings of S and T that have length at most τ.

We run another procedure in parallel in order to compute T-maximal common substrings of length at least τ. During preprocessing, we build the Extended IPM data structure (Theorem 4.3.9) for the string S with $\ell = \tau - 2$ in $O(m\log_{m/\tau} m) = O(nm/\tau)$ time using $O((m/\tau)\cdot\log m(\log\log m)^3) = O(\tau)$ space.

Assume that while reading a window $W = T[\ell..r]$, the sliding-window procedure found an LCSuf $T[i..r]$ of length at least τ. We start a search for a common substring starting in W. Let $j \geq r$ be the current letter of T, and $T[i..j]$, $\ell \leq i \leq r$, be the longest suffix of $T[\ell..j]$ that occurs in S. We assume that we know a position where $T[i..j]$ occurs in S, which is the case for $j = r$. When $T[j+1]$ arrives, we update i using the following observation:

Observation 4.6.3. *If $T[i..j]$ is the longest suffix of $T[1..j]$ that occurs in S, and $T[i'..j+1]$ is the longest suffix of $T[1..j+1]$ that occurs in S, then $i \leq i'$.*

By using binary search and IPM queries, we can find the smallest $i' \geq i$ such that $T[i'..j+1]$ occurs in S and a witness occurrence, if the corresponding string has length at least τ: namely, if $S[x..x']$ is a witness occurrence of $T[i..j]$ in S, we search for occurrences of $P = S[x+(i'-i)..x']T[j+1]$ in S. If $j - i' < \tau$, we stop the search, and otherwise we set

$i' = i$ and continue. It is evident that all T-maximal common substrings of S and T that are of length greater than τ can be extracted during the execution of the above procedure: a maintained suffix of length greater than τ is such a fragment if the last update to it was an increment of its right endpoint, while the next update is an increment of its left endpoint. For every letter, we run at most one binary search which uses $O(\log m)$ IPM queries and hence takes $O(\log m(\log\log m)^3)$ time. As $\tau = O(m/\log^2 m)$, the m/τ term dominates the per-letter running time. The correctness of the described procedure follows from the fact that any substring of T of length greater than τ is either fully contained in the first window or crosses the boundary of some window. □

Corollary 4.6.4. *Assume to be given random access to an m-length string S and streaming access to a n-length string T, where $n \geq m$.*

For all $\tau \in [\sqrt{m}\log m(\log\log m)^3 . . O(m/\log^2 m)]$, there is an algorithm that computes $\mathsf{LCS}(S,T)$ using $O(nm/\tau \cdot \log\log\sigma)$ time and $O(\tau)$ space.

Proof. Note that the longest common substring of S and T is a T-maximal substring of S and T. We use the algorithm of Theorem 4.6.1 with the same value of τ to iterate over all T-maximal common substrings $T[t..t']$ of S and T, and store the pair of indices t, t' that maximizes $t' - t$. □

Corollary 4.6.5. *Assume to be given random access to an m-length pattern P and streaming access to an n-length text T, where $n \geq m$.*

For all $\tau \in [\sqrt{m}\log m(\log\log m)^3 . . O(m/\log^2 m)]$, there is an algorithm that solves the CPM problem for P, T using $O(m/\tau \cdot \log\log\sigma)$ time per letter of T and $O(\tau)$ space.

Proof. We use the algorithm of Theorem 4.6.1 with threshold τ on the string $P \cdot P$, to which we have random access, and a streaming string T. The occurrence of any rotation of P in T implies a common substring of $P \cdot P$ and T of length $m \geq 2\tau$. The algorithm of Theorem 4.6.1 allows us to find such occurrences in $O(m/\tau \cdot \log\log\sigma)$ amortized time per letter of T using $O(\tau)$ space. By noticing that none of the m-length substrings are fully contained in $T(|T| - \tau . . |T|]$, we can deamortise the algorithm using the standard time-slicing technique, cf [104]. □

4.7 CPM in the Read-only Setting

In this section, we present a deterministic read-only online algorithm for the CPM problem.

Theorem 4.7.1. *There is a deterministic read-only online algorithm that solves the CPM problem on a pattern P of length m and a text T of length n using $O(1)$ space and $O(1)$ time per letter of the text.*

Proof. In this proof, we assume that $n \leq 2m - 1$. If this is not the case, we can cover T with $2m$-length windows overlapping by $m - 1$ letters, and process the text window by window; the last window might be shorter. Every occurrence of a rotation of P belongs to exactly one of the windows and hence will be reported exactly once.

We partition P into four fragments P_1, P_2, P_3 and P_4, each of length either $\lfloor m/4 \rfloor$ or $\lceil m/4 \rceil$.[2] By applying Fact 4.2.3, we compute the periods of each of P and P_i for

[2]The sole reason for partitioning P into four fragments instead of two is to guarantee that there is an occurrence of some P_i close to the starting position of each rotation of P. This allows us to obtain a worst-case rather than an amortised time bound for processing each letter of the text.

$i \in [1..4]$, if it is are periodic. We also compute, for each $i \in [1..4]$, the occurrences of P_i in P^2 using Fact 4.2.2, and store them in $O(1)$ space due to Corollary 3.2.2. Overall, the preprocessing step takes $O(m)$ time and uses constant space.

We compute all occurrences of all P_i in T in an online manner using Fact 4.2.2. Due to Corollary 3.2.2, we can represent all computed occurrences of each P_i using a constant number of arithmetic progressions with difference $\mathsf{per}(P_i)$ in $O(1)$ space.

Observation 4.7.2. *Assume that $T(j-m..j] = P[\Delta+1..m] \cdot P[..\Delta]$. There is an occurrence of P_i at a position ℓ of T such that $j-m < \ell \leq j-|P_i|+1$ if and only if there is an occurrence of P_i at position $p = \Delta+\ell-j+m$ of P^2.*

Now, note that for every rotation P' of P, some P_i occurs at one of the first $\phi := 2\lceil m/4 \rceil$ positions of P'. We will use such occurrences as anchors to compute the occurrences of rotations of P in T. Fix i such that there is an occurrence of P_i in the first ϕ positions of $T(j-m..j]$. We consider two cases depending on whether the period of P_i is large or small.

Case I: $\mathsf{per}(P_i) > |P_i|/4$. By Corollary 3.2.2, there are $O(1)$ occurrences of P_i in each of T and P^2. Suppose that P_i occurs at position ℓ of T. If $T(j-m..j] = P[\Delta+1..m] \cdot P[..\Delta]$ for some Δ, then, by Observation 4.7.2, P_i occurs at position $p = \Delta+\ell-j+m$ of P^2 and we must have that the length of the longest common suffix of $T[1..\ell)$ and $P^2[1..p)$ is at least $\ell-(j-m)$ and the length of the longest common prefix of $T[\ell+|P_i|..]$ and $P^2[p+|P_i|..]$ is at least $j-\ell-|P_i|$. As we only need to consider occurrences of P_i in the first ϕ positions of rotations of P, we can work under the assumption that $\ell-(j-m) \leq \phi$. Hence, it suffices to compute, for every occurrence of P_i at a position p in P^2 and every occurrence of P_i at a position ℓ in T, values

- $x := \max\{\phi, \mathsf{LCE}^R(T[1..\ell), P^2[1..p))\}$, the maximum of ϕ and the length of the longest common suffix of $T[1..\ell)$ and $P^2[1..p)$;
- $y := \mathsf{LCE}^R(T[1..\ell), P^2[1..p))$, the length of the longest common prefix of $T[\ell+|P_i|..]$ and $P^2[p+|P_i|..]$.

The length y is computed naively as new letters arrive, while, in order to compute x, we perform a constant number of letter comparisons for each letter that arrives. Since $\ell-(j-m) = O(j-\ell-|P_i|)$, we will have completed the extension to the left when the j-th letter of the text arrives. As there is a constant number of pairs (p,ℓ) to be considered, we perform a total number of $O(1)$ letter comparisons per letter of the text.

Case II: $\mathsf{per}(P_i) \leq |P_i|/4$. For brevity, denote $\rho = \mathsf{per}(P_i)$. Below, when we talk about arithmetic progressions of occurrences of P_i, we mean maximal arithmetic progressions of starting positions of occurrences of P_i with difference ρ. Consider the first element ℓ and the last element r of the rightmost computed arithmetic progression of occurrences of P_i in $T(j-m..j]$. We next distinguish between two cases depending on whether $\mathsf{per}(T(j-m..j]) = \rho$. This information can be easily maintained in $O(1)$ time per letter using $O(1)$ space as follows. In particular, for each arithmetic progression of occurrences of P_i in T, we perform at most $\rho-1$ letter comparisons to extend the periodicity to the left; we can do this lazily upon computing the first element of each progression, by performing at most one letter comparison for each of the next $\rho-1$ letter arrivals. Further, as at most one arithmetic progression corresponds to occurrences of P_i in T that contain a position in $(j-\rho..j]$, the extensions to the right take $O(1)$ time per letter as well.

Subcase (a): $\mathsf{per}(T(j-m..j]) \neq \rho$**.** Suppose that $T(j-m..j] = P[\Delta+1..m] \cdot P[..\Delta]$ for some Δ. Then, due to Observation 4.7.2, one of the two following holds:

1. ℓ and $p_\ell = \Delta+\ell-j+m$ are the first elements in arithmetic progressions of occurrences of P_i in $T(j-m..j]$ and P^2, respectively;

2. r and $p_r = \Delta + r - j + m$ are the last elements in arithmetic progressions of occurrences of P_i in $T(j-m..j]$ and P^2, respectively.

We handle this case by considering a subset of pairs of occurrences of P_i and treating them similarly to Case I. Namely, we consider (a) pairs that are first in their respective arithmetic progressions in P^2 and T and (b) pairs that are last in their respective arithmetic progressions in P^2 and $T(j-m..j]$. By Corollary 3.2.2, there are only a constant number of such elements in P^2 and a constant number of such elements in the text at any time (a previously last element in the text may stop being last when a new occurrence of P_i is detected). For pairs of first elements there are no changes required to the algorithm for Case I. We next argue that, for each pair (r, p_r) of last elements, it suffices to perform only $O(\rho)$ letter comparisons to check how far the periodicity extends to the left, and that this is all we need to check. Due to this, we do not restrict our attention to the case when $r \in (j-m..j-m+\phi]$, but rather consider all last elements of arithmetic progressions. Let ℓ' be the first element of the arithmetic progression in $T(j-m..m]$ that contains r. If $\ell' > \rho + j - m$, we avoid extending to the left since either $\ell' \in (j-m..j-m+\phi]$ and the sought occurrence of a rotation of P, if it exists, will be computed by the algorithm when it processes pair $(\ell', \Delta + \ell' - j + m)$ or the sought occurrence will be computed when processing a different arithmetic progression of occurrences of P_i or a different P_j. Further note that the extension to the left has been already computed; either ℓ' is not the first element in the arithmetic progression of occurrences of P_i in T (we have assumed that it is in $T(j-m..j]$), in which case we are trivially done, or ℓ' is the first element of an arithmetic progression in T and hence we extended the periodicity via a lazy computation when the occurrence of P_i at position ℓ' was detected. As the occurrences of P_i in T are spaced at least ρ positions away, the above procedure takes $O(1)$ time per letter of the text.

Subcase (b): $\mathsf{per}(T(j-m..j]) = \rho$. Using $O(m)$ time and $O(1)$ extra space, we can precompute all $1 \le j \le \rho$ such that $Q_i^{\infty}[j..j+m)$ occurs in P^2, where $Q_i = P_i[1..\rho]$; it suffices to extend the periodicity for each of the $O(1)$ arithmetic progressions of occurrences of P_i in P^2 and to perform standard arithmetic. In particular, the output consists of a constant number of intervals. Then, if $\mathsf{per}(T(j-m..j]) = \rho$, $T(j-m..j]$ equals a rotation of P if and only if $\ell - (j-m) \pmod{\rho}$ is in one of the computed intervals and this can be checked in constant time. □

Chapter 5

Pattern Matching with Mismatches and Wildcards

5.1 Introduction

Pattern matching is one of the most fundamental algorithmic problems on strings. Given a text T of length n and a pattern P of length m, both over an alphabet Σ, the goal is to compute all occurrences of P in T. This problem admits an efficient linear-time solution such as the seminal algorithm of Knuth, Morris, and Pratt [201]. However, looking for exact matches of P in T can be too restrictive for some applications, for example when working with potentially corrupted data, when accounting for mutations in genomic data, or when searching for an incomplete pattern. There are several ways to model and to work with corrupt or partial textual data for the purposes of pattern matching.

A natural and well-studied problem is that of computing the similarity between fragments of the text and a given pattern, with respect to some distance metric. One of the most commonly used such metrics is the Hamming distance. Abrahamson [14] and Kosaraju [213] independently developed an $O(n\sqrt{m \log m})$-time algorithm that computes the Hamming distance between the pattern and every m-length substring of the text using convolutions via the Fast Fourier Transform (FFT). This complexity has only been recently improved with the state of the art being the randomised $O(n\sqrt{m})$-time algorithm of Chan et al. [86] and the deterministic $O(n\sqrt{m \log \log m})$-time algorithm of Jin and Xu [187].

In many applications, one is interested in computing substrings of the text that are *close* to the pattern instead of computing the distance to every substring. In this case, an integer threshold k is given as part of the input and the goal is to compute fragments of T that have at most k mismatches with P. Such a fragment is called a *k-mismatch occurrence* of P in T. The state-of-the-art for this problem are the $O(n\sqrt{k \log k})$-time algorithm of Amir et al. [32], the $O(n + (n/m) \cdot k^2)$-time algorithm of Chan et al. [85], and the $O(n + kn/\sqrt{m})$-time algorithm of Chan et al. [86] that provides a smooth trade-off between the two aforementioned solutions, improving the bound for some range of parameters. Deterministic counterparts of the last two algorithms (which are randomised) at the expense of extra polylogarithmic factors were presented in [89, 105, 158].

The structure of the set of k-mismatch occurrences of P in T admits an insightful characterisation, shown by Charalampopoulos et al. [89] who tightened the result of Bringmann et al. [80]: either P has $O(k \cdot n/m)$ k-mismatch occurrences in T or P is at Hamming distance less than $2k$ from a string with period $q = O(m/k)$; further, in the periodic case, the starting positions of the k-mismatch occurrences of P in T can be parti-

tioned into $O(k^2 \cdot n/m)$ arithmetic progressions with common difference q. This characterization can be exploited towards obtaining efficient algorithms in settings other than the standard one, e.g., in the setting where both P and T are given in compressed form [89], and, in combination with other ideas and techniques, in the quantum setting [186], in the online read-only setting [49], and in the differentially private setting [274].

In the case when the positions of the corrupt characters in the two strings are known in advance, one can use a more adaptive approach, by placing a *wildcard* $\lozenge \notin \Sigma$, a special character that matches any character in $\Sigma \cup \{\lozenge\}$, in each of these positions, and then performing exact pattern matching. Already in 1974, Fischer and Paterson [140] presented an $O(n \log m \log \sigma)$-time algorithm for the pattern matching problem with wildcards. Subsequent works by Indyk [184], Kalai [189], and Cole and Hariharan [111] culminated in an $O(n \log m)$-time deterministic algorithm [111]. A few years later, Clifford and Clifford [100] presented a very elegant algorithm with the same complexities. All the above solutions are based on fast convolutions via the FFT.

Pattern matching with mismatches and wildcards. Unsurprisingly, the pattern matching problem in the case when we both have wildcards and allow for mismatches has also received significant attention. Conceptually, it covers the case when some of the corrupt positions are known, but not all of them. We denote by D the total number of wildcards in P and T, and by G the number of maximal fragments in P and T all of whose characters are wildcards. A summary of known results for the considered problem is provided in Table 5.1. Note that, as discussed below, any algorithm for the pattern matching problem with at most k mismatches (without wildcards) can be applied to the setting where we have wildcards only in the pattern at the expense of allowing for $k + D$ mismatches instead of k (and hence replacing any factor of k in the complexity by a factor $k + D$). We chose to not include the implied results in the table to avoid clutter.

Note that, in practice, G may be much smaller than D. For example, DNA sequences have biologically important loci, which are characterised using the notion of structured motifs [238]: sequences of alternating conserved and non-conserved blocks. Conserved blocks are ones which are identical across intra- or inter-species occurrences of the structured motif, while non-conserved ones are not known to have biological significance and can vary significantly across such occurrences. Non-conserved blocks can be hence modelled with blocks of wildcards as in [235]. In this case, evidently, we have G being much smaller than D. This feature has been used in the literature before, e.g., for the problem of answering longest common compatible prefix queries over a string with wildcards. Crochemore et al. [116] showed an $O(nG)$-time construction algorithm for a data structure that is capable of answering such queries in $O(1)$ time, while the previously best known construction time was $O(nD)$ [70].

In several applications, it is sufficient to only account for wildcards in one of P and T: in the application we just discussed, the text is a fixed DNA sequence, whereas the sought pattern, the structured motif, is modelled as a string with wildcards. In such cases, one can obtain more efficient solutions than those for the general case when both P and T have wildcards, such as the ones presented in [101, 245] and the one we present here.

In this chapter, we describe our algorithms in the `PILLAR` model, introduced by Charalampopoulos et al. [89]: see Section 3.3 for a description of this framework. We will make use of the "standard trick" of pattern matching (see Section 3.2.1), which allows us to assume that the length n of the text T is at most $3m/2$. An algorithm with runtime $C(m)$ for this case implies an algorithm with runtime $O(C(m) \cdot n/m)$ for the general case.

Table 5.1: Results on pattern matching with wildcards under the Hamming distance.

Time complexity	$\Diamond$ in	Reference
$O(nk^2 \log^2 m)$	P and T	[103]
$O(n(k + \log m \log k) \log n)$		[103]
$O(nk \operatorname{polylog} m)$		[102, 103, 244]
$O(n\sqrt{m \log m})$		[14, 213], via. [103]
$O(n\sqrt{m - D} \log m)$		[32]
$O(n\sqrt[3]{mk \log^2 m})$	one of P or T	[101]
$O(n(\sqrt{k \log m} + \min\{\sqrt[3]{Gk \log^2 m}, \sqrt{G \log m}\}))$	P	[245]
$\tilde{O}(n\sqrt{k})$		[243]
$O(n + (n/m)(D + k)(G + k))$		**this work**

Reduction to pattern matching with mismatches.

The problem of k-mismatch pattern matching with D wildcards can be straightforwardly reduced to $(D + k)$-mismatch pattern matching in *solid strings*, i.e., strings without wildcards. In what follows we only consider solid texts, i.e. we assume that T does not contain any wildcards. Given the pattern P, construct the string $P_\#$ obtained by replacing every wildcard in P with a new character $\# \notin \Sigma$. Observe that a pattern P with D wildcards has a k-mismatch occurrence at a position i of a solid text T if and only if $P_\#$ has a $(D + k)$-mismatch occurrence at that position.

In [89], the authors present an efficient algorithm for the d-mismatch pattern matching problem for solid strings in the `PILLAR` model.

Theorem 5.1.1 ([89, Main Theorem 8])**.** *Let S and T be solid strings of respective lengths m and $n \leq 3m/2$. We can compute a representation of the d-mismatch occurrences of S in T using $O(d^2 \log \log d)$ time plus $O(d^2)$ `PILLAR` operations.*

Applying Theorem 5.1.1 with $S = P_\#$ and $d = D + k$, we obtain an algorithm for k-mismatch pattern matching with D wildcards that runs in $\tilde{O}((D + k)^2)$ time in the `PILLAR` model.

Our results.

We provide a more fine-grained result, replacing one D factor with a G factor. We also make an analogous improvement over the structural result for the set of k-mismatch occurrences obtained via the reduction to $(D+k)$-mismatch pattern matching. Our main result can be formally stated as follows.

Theorem 5.1.2. *Let P be a pattern of length m with D wildcards arranged in G groups, T be a solid text of length $n \leq 3m/2$, and k be a positive integer. We can compute a representation of the k-mismatch occurrences of P in T as $O((D + k)G)$ arithmetic progressions with common difference and $O((D+k)k)$ additional occurrences using $O((D+k) \cdot (G + k) \log \log(D + k))$ time plus $O((D + k) \cdot (G + k))$ `PILLAR` operations.*

In the usual word RAM model, by using known implementations of the PILLAR operations with $O(n)$ preprocessing time and $O(1)$ operation time, the "standard trick", and observing that the loglogarithmic factor can be avoided at the cost of $O(n)$ extra time, we obtain an algorithm with runtime $O(n + (n/m)(D + k)(G + k))$ for texts of arbitrary length n. Section 5.5 details the implementation of our algorithm in other settings, such as the dynamic and compressed settings. For example, given a solid text T and a pattern P with D wildcards represented as straight-line programs of sizes N and M respectively, we can compute the number of k-mismatch occurrences of P in T in time $\tilde{O}(M + N \cdot (D + k)(G + k))$, without having to uncompress P and T.

We complement our structural result with a lower bound on the number of arithmetic progressions of occurrences of a pattern with mismatches and wildcards (Theorem 5.6.1), based on a neat construction that employs large sets that do not contain any arithmetic progression of size 3 [56, 127, 129]. Informally, we show that there exist a pattern P and a text T of length at most $3|P|/2$ such that the set of k-mismatch occurrences of P in T cannot be covered with less than $\Omega((D+k)\cdot(k+1))$ arithmetic progressions. This implies, in particular, a lower bound of $\Omega(D)$ on the number of arithmetic progressions of exact occurrences for a pattern with D wildcards and a lower bound of $\Omega(k^2)$ on the number of arithmetic progressions of k-mismatch occurrences of a solid pattern, thus showing the tightness of the known upper bound [89].

When $k = 0$, Theorem 5.1.2 readily implies an $O(n + DG \cdot n/m)$-time algorithm for *exact* pattern matching. However, the techniques employed by this algorithm are rather heavy-handed, and this can be avoided. In Section 5.3, we present a much simpler algorithm that achieves the same time complexity and showcases the primary technical innovation of our approach: the utilization of carefully selected positions, termed *sparsifiers*, which exclusively belong to fragments F of P such that the ratio of the number of wildcards within them to their length is bounded by $O(D/m)$. In the standard word RAM model, the implied $O(n + DG \cdot n/m)$ time complexity for exact pattern matching outperforms the state-of-the-art $O(n \log m)$ [100, 111] when $DG = o(m \log m)$.

5.1.1 Technical Overview

To illustrate how sparsifiers help, consider our algorithm for exact pattern matching, which draws ideas from the work of Bringmann et al. [80]. We first compute a solid $\Omega(m/G)$-length fragment S of P that contains a sparsifier. We then compute its exact matches in T. If S only has a few occurrences, we straightforwardly verify which of those extend to occurrences of P. However, if S has many occurrences we cannot afford to do that, and we instead have to exploit the implied periodic structure of S. We distinguish between two cases. In the case when P matches a periodic string with the same period as S, denoted $\mathsf{per}(S)$, we take a sliding window approach as in [80], using the fact that the wildcards are organised in only G groups. The remaining case poses the main technical challenge. In that case, our goal is to align the maximal fragment $S' := P[i..j]$ of P that contains S and matches a solid string with period $\mathsf{per}(S)$ with a periodic fragment of T such that position $i-1$ is aligned with a position breaking the periodicity in T; a so-called *misperiod*. To this end, we compute $O(G)$ maximal fragments of T, called S-runs, that have period $\mathsf{per}(S)$. The issue is, however, that up to D misperiods in T might be aligned with wildcards of S'. A straightforward approach would be to extend each S-run to the left, allowing for $D+1$ misperiods and to try aligning each such misperiod with $i-1$. This would yield an algorithm with runtime $O(G^2D)$ in the PILLAR model, as we would have $O(DG)$ candidate misperiods to align position $i-1$ with, and the verification time for

each such alignment is $O(G)$. The crucial observation is that since S' contains a sparsifier, we do not need to extend each S-run allowing for $D+1$ misperiods. Instead, we extend it while the ratio of the encountered misperiods to its length does not exceed $20 \cdot D/m$. By skipping S-runs that are covered due to the extension of other S-runs, we ensure that the total number of misperiods with which we align $i-1$ is only $O(D)$, obtaining the desired complexity.

As for handling mismatches, we follow the framework of Charalampopoulos et al. [89] for k-mismatch pattern matching on solid strings. They showed that an efficient structural analysis of a solid pattern can return a number of so-called *breaks* or a number of so-called *repetitive regions*, or conclude that the pattern is *almost periodic*. They treated each of the three cases separately, exploiting the computed structure. We make several alterations to account for wildcards, such as ensuring that breaks are solid strings and adapting the sliding window approach. The primary technical challenge in achieving an efficient solution lies in limiting the number of occurrences of repetitive regions in T. The greater the number of repetitive region occurrences, the higher the number of potential starting positions for k-mismatch occurrences of P. We achieve that by ensuring that each repetitive region contains a sparsifier. This way, we force an upper bound on the number of wildcards in each repetitive region, which, in turn, allows us to bound the number of its approximate occurrences in T.

Our lower bound is based on a neat construction that employs large sets that do not contain any arithmetic progression of size 3 [56, 127, 129]. We use these sets to construct the pattern P and the text T. They consist mostly of 0s, except that P contains wildcards and 1s positioned at indices that form a progression-free set, and T contains 0s also positioned at indices that form a progression-free set, but that are far apart from each other. Our construction ensures that there is a k-mismatch occurrence of P at position i in T if and only if a 1 or a wildcard of P is aligned with a 1 of T. We show that the set of such positions i has size $\Omega((D+k)\cdot(k+1))$ and is progression-free.

Other Related Work

The pattern matching problem with wildcards under the edit distance has also been studied. A straightforward extension of the $O(nk)$-time algorithm of Landau and Vishkin [227] for pattern matching under edit distance for solid P and T yields an algorithm with running time $O(n(k+G))$. Later, Akutsu [20] presented an algorithm running in time $O(n\sqrt{mk}\operatorname{polylog} m)$. Recently, an algorithm with runtime $O(n(k+\sqrt{Gk\log n}))$ was presented [50], improving over both previously known algorithms when $k \ll G \ll m$. The aforementioned algorithms can handle the case when both P and T contain wildcards.

Organisation of the Chapter

Section 5.2 introduces concepts relevant to this work, as well as an abstract problem that we use in our analysis. Section 5.3 presents an algorithm for exact pattern matching with wildcards in the `PILLAR` model. It illustrates some of the main ideas underlying this work, without having to handle mismatches, which bring further challenges. In Section 5.4, we describe the algorithm that underlies Theorem 5.1.2; the implications of this theorem in different settings are provided in Section 5.5. We conclude with Section 5.6 where we present our structural lower bound.

5.2 Preliminaries

In this work, Σ denotes an alphabet that consists of integers polynomially bounded in the length of the input strings. The elements of Σ are called *(solid) characters*. Additionally, we consider a special character denoted by $\diamond$ that is not in Σ and is called a *wildcard*. Let $\Sigma_\diamond = \Sigma \cup \{\diamond\}$. We say that two characters *match* if they are identical or at least one of them is a wildcard. Two equal-length strings match if and only if their i-th characters match for all i.

In this section, we extend to strings over $\Sigma_\diamond$ the notations introduced in Section 2.1, e.g. if $T \in \Sigma_\diamond^n$, then for $i \leq j$, $T[i..j]$ denotes the substring $T[i]T[i+1]\dots T[j]$ of T. A string in $(\Sigma_\diamond)^*$ is called *solid* if it only contains solid characters, i.e., it is in Σ^*. Additionally, we define the *ball* $B_T(i,r)$ with a radius r and a center i in T, as the fragment $T[\max\{1, i-r\}..\min\{i+r, n\}]$, where we often omit the subscript T if it is clear from the context.

An integer ρ is a *deterministic period* of a string $S \in \Sigma_\diamond^*$ (that may contain wildcards), if there exists a solid string T that matches S and has period ρ.

The Hamming distance $\delta_H(S_1, S_2)$ between two equal-length strings S_1, S_2 in $\Sigma_\diamond^*$ is the number of positions i such that $S_1[i]$ does not match $S_2[i]$. For two strings $U, Q \in \Sigma_\diamond^*$, we slightly abuse notation and denote $\delta_H(U, Q^\infty[..|U|])$ by $\delta_H(U, Q^\infty)$. A position i of a string T is called a k-mismatch occurrence of a string P if $\delta_H(T[i, i+|P|), P) \leq k$, and the set of all k-mismatch occurrences of P in T is denoted by $\mathrm{Occ}_k(P, T)$.

5.2.1 The PILLAR Model

In this chapter, we describe our algorithms in the PILLAR model, introduced by Charalampopoulos et al. [89]: see Section 3.3 for a description of this framework. We will use the following operation, which can be efficiently implemented in the PILLAR model.

Fact 5.2.1 ([92, proof of Lemma 12])**.** *For a fragment X and a suffix Z of a solid string Y, the value $\mathsf{LCP}(X^\infty, Z)$ can be computed using $O(1)$ PILLAR operations.*

To work in the PILLAR model despite considering strings with wildcards, we replace each wildcard with a solid character $\# \notin \Sigma$ and using PILLAR operations over the obtained collection of (solid) strings. In our algorithms, we will use PILLAR operations in the solid part of the input strings, and use properties of wildcards to handle the other parts. Namely, for each string in the collection, we pre-compute a linked list that stores the endpoints of groups of wildcards. One example of algorithm that alternates between PILLAR operations and wildcard groups is the Kangaroo jumping technique of Landau and Vishkin [224].

Fact 5.2.2. *Let P be a pattern with D wildcards arranged in G groups and T be a solid text. For a position p and a given threshold $k \geq 0$, one can test whether $\delta_H(P, T[p..p+m)) \leq k$ using $O(G+k)$ PILLAR operations.*

Proof. We start at position 1 of the pattern with mismatch budget k. With one LCP query, we reach the first position j in the pattern such that either $P[j] = \diamond$ or $P[j] \neq T[p+j-1]$. If $P[j] = \diamond$, we jump to the next non-wildcard character using another LCP query (using Fact 5.2.1); otherwise, we decrement the budget of mismatches by one and continue from the next position. If the budget of mismatches becomes zero before we reach the end of P, then $\delta_H(P, T[p..p+m)) > k$; otherwise it is $\leq k$. In each iteration we either decrease the mismatch budget or jump over a group of wildcards. Hence, the total number of performed LCP queries (which are the bottleneck) is $O(G+k)$. □

5.2.2 Sparsifiers

In Section 5.1, we elucidated the pivotal role of the fragments of the pattern where wildcards exhibit a "typical" distribution. In this section, we formalize this concept.

Definition 5.2.3 (Sparsifiers)**.** *Consider a string $X \in \Sigma_{\diamond}^{m}$ containing D wildcards. We call a position i in X a* sparsifier *if $X[i]$ is a solid character and, for any r, the number of wildcards within the ball of radius r centered at i is at most $8r \cdot D/m$.*

In the following, we demonstrate that P contains a long fragment whose every position is a sparsifier. We start with an abstract lemma, where one can think of a binary vector V as the indicator vector for wildcards, and $\|V\|$ denotes the number of 1s in V. A *run of 1s (resp. 0s)* is a maximal fragment that consists only of 1s (resp. 0s).

Lemma 5.2.4. *Let V be a binary vector of size N, $M := \|V\|$ and R be the number of runs of 1s in V. Assume V to be represented as a linked list of the endpoints of runs of 1s in V, arranged in the sorted order. There is an $O(R)$-time algorithm that computes a set $U \subseteq [1..N]$, represented as the union of at most $R+1$ disjoint intervals, such that:*

1. *$|U| \geq N/2 - M$,*
2. *for each $i \in U$ and radius $r \in [1..N]$, $\|B_V(i,r)\| \leq 8r \cdot M/N$.*

Proof. First, we scan V from left to right. Each time we see a 1, we mark the $N/(4M)$ leftmost 0s that are to the right of the considered 1 and have not been already marked. If we mark the last 0 in V, we terminate the scan. Overall, we mark at most $N/4$ 0s. Next, we perform the symmetric procedure in a right-to-left scan of V. Let U be the set of positions of unmarked 0s after these two marking steps. Observe that, by construction, every run of 0s contains at most one interval of unmarked positions and hence U can be represented as union of at most $R+1$ disjoint intervals.

Let us show that U satisfies the condition of the lemma. First, we have $|U| \geq N - M - N/2 \geq N/2 - M$. Now, fix $i \in U$ and $r \in [1..N]$. Let $B_1 = V[\max\{i-r,1\}..i-1]$ and $B_2 = V[i+1..\min\{i+r,N\}]$. Since $V[i]$ was not marked in the left-to-right scan, there are at least $\|B_1\| \cdot N/(4M)$ 0s in B_1. Symmetrically, since $V[i]$ was not marked in the right-to-left scan of B, there are at least $\|B_2\| \cdot N/(4M)$ 0s in B_2. On the other hand, the number of 0s in B_1 (resp. B_2) is bounded by r, and therefore

$$\|B_V[i,r]\| = \|B_1\| + \|B_2\| \leq 4r \cdot M/N + 4r \cdot M/N \leq 8r \cdot 4M/N.$$

It remains to show that the algorithm can be implemented efficiently. In addition to the linked list L_1 representing the runs of 1s in V, the algorithm maintains a linked list L_2 of the intervals of marked 0s, sorted by their left endpoints. The algorithm simulates marking the 0s for all 1s in the current run at once, taking $O(R)$ time in total. Having computed L_2, the algorithm scans L_1 and L_2 in parallel in $O(R)$ time to extract the set U, represented as the union of at most $R+1$ disjoint intervals. This completes the proof of the lemma. □

An application of Lemma 5.2.4 to P with wildcards treated as 1s and solid characters treated as 0s yields the following corollary.

Corollary 5.2.5. *Consider a string $P \in \Sigma_{\diamond}^{m}$ containing D wildcards arranged in G groups. Given the endpoints of those groups, in $O(G)$ time, we can compute a set of sparsifiers of size at least $m/2 - D$ represented as the union of at most $G+1$ disjoint intervals.*

5.3 Exact Pattern Matching in the PILLAR Model

In this section, we consider a pattern P of length m with $D \geq 1$ wildcards arranged in G groups and a solid text T of length n such that $n \leq 3m/2$. We then use the "standard trick" presented in the introduction to lift the result to texts of arbitrary length. We prove a structural result for the exact occurrences of P in T and show how to compute them efficiently when the product of D and G is small. In particular, we compute them in linear time when $DG = O(m)$, thus improving by a logarithmic factor over the state-of-the-art $O(n \log m)$-time algorithms in this case.

Definition 5.3.1 (Misperiods)**.** *Consider a string V over alphabet $\Sigma_\Diamond$. We say that a position x is a misperiod with respect to a solid fragment $V[i..j]$ when $V[x]$ does not match $V[y]$, where y is any position in $[i..j]$ such that* $\mathsf{per}(V[i..j])$ *divides $|y-x|$. Additionally, we consider positions 0 and $|V|+1$ as misperiods. We denote the set of the at most κ rightmost misperiods smaller than i with respect to $V[i..j]$ by* $\mathsf{LeftMisper}(V, i, j, \kappa)$*. Similarly, we denote the set of the at most κ leftmost misperiods larger than j with respect to $V[i..j]$ by* $\mathsf{RightMisper}(V, i, j, \kappa)$*.*

Example 5.3.2. Consider a string $V = \texttt{cc}\Diamond\texttt{bdabcabcabcab}$. The misperiods with respect to the underlined and highlighted fragment $V[6..13]$, which has period 3, are positions 0, 1, 5, and $|V|+1 = 17$. We have $\mathsf{LeftMisper}(V, 6, 13, 2) = \{1, 5\}$ and $\mathsf{RightMisper}(V, 6, 13, 2) = \{17\}$.

The next lemma states that misperiods can be computed efficiently in an incremental fashion. Its proof uses the kangaroo method, and closely follows [80, 92].

Lemma 5.3.3. *Consider a string V over an alphabet $\Sigma_\Diamond$ and a solid periodic fragment $V[i..j]$ of V. The elements of either of* $\mathsf{LeftMisper}(V, i, j, |V|)$ *and* $\mathsf{RightMisper}(V, i, j, |V|)$ *can be computed in the increasing order with respect to their distance from position i so that:*

- *the first misperiod x can be computed in $O(1+G_0)$ time in the PILLAR model, where G_0 denotes the number of groups of wildcards between positions x and i;*
- *given the t-th misperiod $x \notin \{0, |V|+1\}$, the $(t+1)$-th misperiod can be computed in $O(1+G_t)$ time in the PILLAR model, where G_t denotes the number of groups of wildcards between said misperiods.*

Proof. It suffices to describe how to compute elements of $\mathsf{RightMisper}(V, i, j, |V|)$ as the computation of elements of $\mathsf{LeftMisper}(V, i, j, |V|)$ is symmetric.

We first compute the period q of $V[i..j]$. Due to our assumption that $V[i..j]$ is periodic, this can be done in $O(1)$ time in the PILLAR model [208]. Let $Q := V[i..i+q)$ and observe that Q^2 is a prefix of $V[i..j]$ since the latter is periodic. Hence, in constant time, we can retrieve a fragment of V equal to any desired rotation of Q.

Let us now discuss how to efficiently compute the first misperiod, that is, the first mismatch between $V[i..|V|]$ and Q^∞. Let F be the maximal solid fragment that contains $V[i..j]$. Using Fact 5.2.1, in $O(1)$ time in the PILLAR model, we either compute the first misperiod or reach a group of wildcards and conclude that there are no misperiods in F. In the latter case we do the following. While we have not yet found a misperiod or reached the end of V, we consider the subsequent maximal solid fragment Y; we find the starting position y of this fragment by applying Fact 5.2.1 to compute the length of the encountered group of wildcards. The elements of $\mathsf{RightMisper}(V, i, j, |V|)$ spanned by Y

are the mismatches of Y and $Q^\infty[y-i..y-i+|Y|)$. We can compute the smallest such element, if one exists, using Fact 5.2.1, or reach the next group of wildcards.

Subsequent misperiods are computed in an analogous manner using the kangaroo method: we try to compute them in maximal solid fragments using Fact 5.2.1 in $O(1)$ time in the PILLAR model, while skipping any encountered groups of wildcards in $O(1)$ time in the PILLAR model. □

A direct application of the above lemma yields the following fact.

Corollary 5.3.4. *For any $k \geq 0$, the sets $\mathsf{LeftMisper}(V,i,j,k)$ and $\mathsf{RightMisper}(V,i,j,k)$ can be computed in $O(k+G)$ time in the PILLAR model.*

Definition 5.3.5. *For two strings S and Q, let $\mathsf{MI}(S,Q)$ denote the set of positions of mismatches between S and Q^∞.*

Definition 5.3.6 (S-runs)**.** *A fragment of a solid string V spanned by a set of occurrences of a solid string S in V whose starting positions form a maximal arithmetic progression with difference $\mathsf{per}(S)$ is called an S-run.*

Example 5.3.7. Let $V = \texttt{cababcabcabcabc}$ and $S = \texttt{abcab}$. The underlined and highlighted fragment $V[4..14]$ is the sole S-run in V; it is spanned by the occurrences of S at positions 4, 7, and 10.

The following fact characterises the overlaps of S-runs.

Fact 5.3.8. *Two S-runs can overlap by no more than $\mathsf{per}(S)-1$ positions.*

Proof. Towards a contradiction suppose that we have two S-runs R_1 and R_2 that overlap by at least $\mathsf{per}(S)$ positions. Since S-runs correspond to inclusion-maximal arithmetic progressions of occurrences of S, the difference of the starting positions of R_1 and R_2 is not a multiple of $\mathsf{per}(S)$. Without loss of generality, suppose that the starting position of R_1 is to the left of the starting position of R_2. Then, $S[1..\mathsf{per}(S)] = R_2[1..\mathsf{per}(S)]$ matches a non-trivial rotation of itself, and is hence not primitive, contradicting Fact 2.1.2. □

We need a final ingredient before we prove the main theorem of this section. We state a more general variant of the statement than we need here that also accounts for k mismatches, as this will come handy in the subsequent section. For the purposes of this section one can think of k as 0. The following corollary follows from [89, Lemma 4.6] via the reduction to computing $(D+k)$-mismatch occurrences of $P_\#$ in T.

Corollary 5.3.9 (of [89, Lemma 4.6])**.** *Let S be a string of length m with D wildcards, let T be a solid string such that $|T| \leq 3|S|/2$, let $k \in [0..m]$ and $d \geq 2(D+k)$ be a positive integer, and let Q be a primitive solid string such that $|Q| \leq m/8d$ and $\delta_H(S,Q^\infty) \leq d$. Then, we can compute, in $O(d)$ time in the PILLAR model, a fragment $T' = T[\ell..r]$ of T such that*

- *$\delta_H(T',Q^\infty) \leq 3d$, and*
- *all elements of $\mathrm{Occ}_k(S,T') = \{p-\ell : p \in \mathrm{Occ}_k(S,T)\}$ are equal to $0 \pmod{|Q|}$.*

Theorem 5.3.10. *Consider a pattern P of length m with D wildcards arranged in G groups and a solid text T of length $n \leq 3m/2$. Then at least one of the following holds:*

- *P has $O(D)$ occurrences in T, or*
- *P has a deterministic period $q = O(m/D)$.*

Furthermore, a representation of the occurrences of P in T can be computed using $O(DG)$ PILLAR operations plus $O(DG \log\log D)$ time. In the former case the occurrences are returned explicitly, while in the latter case they are returned as $O(DG)$ arithmetic progressions with common difference q.

Proof. First, observe that if $D = \Theta(m)$ the statement holds trivially as there can only be $O(m)$ occurrences and we can compute them using $O(mG)$ PILLAR operations, e.g., by applying Fact 5.2.2 for each position of the text. We thus henceforth assume that $D < m/4$.

We apply Corollary 5.2.5 to P, thus obtaining, in $O(G)$ time, $m/2 - D > m/4$ sparsifiers in the form of $G+1$ intervals. Among the output, there exists at least one interval J that is of size at least $L := \lfloor m/(8G) \rfloor \leq m/(4(G+1))$. Let x, y be such that $[x..y] \subseteq J$ is of size L. We set $S := P[x..y]$, noting that S is a solid fragment. Then, we compute all occurrences of S in T in $O(G)$ time in the PILLAR model, represented as $O(G)$ arithmetic progressions with common difference $\mathsf{per}(S)$ (see Fact 3.2.1).

Case (I): S has less than $384D$ occurrences in T. In this case, we try to extend each such occurrence to an occurrence of P in T using Fact 5.2.2 in $O(G)$ time in the PILLAR model. This takes $O(DG)$ time in total in the PILLAR model.

Case (II): S has at least $384D$ occurrences in T. In this case, we have two occurrences of S in T starting within $(3m/2)/(384D)$ positions of each other, and hence $\mathsf{per}(S) \leq m/(256D)$. Let $Q = S[1..\mathsf{per}(S)]$. By definition of $\mathsf{per}(S)$, S is a prefix of Q^∞ and by Fact 2.1.2, Q is primitive. Using Lemma 5.3.4, we compute the sets $\mathsf{LeftMisper}(P, x, y, 1)$ and $\mathsf{RightMisper}(P, x, y, 1)$ in $O(G)$ time in the PILLAR model. In other words, we compute the maximal fragment V of P that contains S and matches exactly some substring of Q^∞.

Subcase (a): $V = P$. We conclude that $q := |Q| \leq m/(256D)$ is a deterministic period of P. We replace Q by its (possibly trivial) rotation Q_0 such that P is equal to a prefix of Q_0^∞ and then apply Corollary 5.3.9 to P and T with $d = 32D$ and $k = 0$ to compute, in $O(D)$ time in the PILLAR model, a fragment T' of T that contains the same number of occurrences of P as T, is at Hamming distance $O(D)$ from a prefix of Q^∞, and only has occurrences of P at positions equivalent to $1 \pmod q$.

It now suffices to show how to compute the occurrences of P in T'. As in previous works [80, 89], we take a sliding window approach. Let W be the set of positions in P where we have a wildcard. For $i \in [1..|T'| - m + 1]$, define $\mathsf{Hidden}(i)$ to be the size of the intersection of $\mathsf{MI}(T', Q) \cap [i..i+m)$ with $i + W$.[1] Intuitively, this is the number of mismatches between $T'[i..i+m)$ and Q^∞ that are aligned with a wildcard in P (and are hence "hidden") when we align P with $T'[i..i+m)$. $\mathsf{Hidden}(\cdot)$ is a step function whose value changes $O(DG)$ times as we increase i, since each mismatch enters or exits the window $[i..i+m)$ at most once and whether it is hidden or not changes at most $2G$ times. We compute $\mathsf{Hidden}(1)$ and store the positions where the function changes (as well as by how much) as events in the increasing order; this sorting takes $O(DG \log\log D)$ time [170].

For a position $i \leq |T'| - m + 1$ with $i \equiv 1 \pmod q$, we have

$$\begin{aligned} d_i &:= \delta_H(T'[i..i+m), P) \\ &= \mathsf{MI}(T'[i..i+m), Q) - \mathsf{Hidden}(i). \end{aligned}$$

[1] For a set Y and an integer z, by $z + Y$ we denote the set $\{z + y : y \in Y\}$.

We maintain this value as we, intuitively, slide P along T, q positions at a time. If there are no events in $(i..i+q] \subseteq [1..|T'|]$, then $d_i = d_{i+q}$. This allows us to report all occurrences of P in T efficiently as $O(DG)$ arithmetic progressions with common difference q by processing all events in a left-to-right manner in $O(DG)$ time.

Subcase (b): $V \neq P$**.** Our goal is to show that, in this case, the occurrences of P in T are $O(D)$ and they can be computed in time $O(GD)$. Without loss of generality, assume that V is not a prefix of P. This means that $\mathsf{LeftMisper}(P, x, y, 1) = \{\mu\} \neq \{0\}$. The occurrences of S in T give us a collection $\mathcal{S}$ of $O(G)$ S-runs in T, any two of which can overlap by less than $\mathsf{per}(S) = q$ positions due to Fact 5.3.8. For each S-run R, extend R to the left until either of the following two conditions is satisfied, keeping track of the encountered misperiods:

(a) the ratio of encountered misperiods to the sum of $|R|$ and the number of prepended positions exceeds $20D/m$,
(b) the beginning of T has been reached.

Denote by E_R the resulting fragment of T and by $\mathcal{M}_R$ the set of misperiods in it. The following two claims are of crucial importance for the algorithm's performance.

▷ Claim 5.3.11. If $p+1$ is an occurrence of P in T that aligns S with an occurrence of S in an S-run $R = T[r..r']$, then $p + \mu \in \mathcal{M}_R$.

Proof. We first show that $p+\mu \in \mathsf{LeftMisper}(T, r, r', |T|)$. We have that the solid character $P[\mu]$ is different from a character $P[\pi]$, where $\pi \in [x..y]$ and $\pi - \mu \equiv 0 \pmod{q}$. Further, $P[\pi] = T[p+\pi]$, where $p+\pi \in [r..r']$, and $P[\mu] = T[p+\mu]$. This means that $T[p+\mu] = P[\mu] \neq P[\pi] = T[p+\pi]$. Now, since $(p+\pi) - (p+\mu) = \pi - \mu$ is divisible by q, which is the period of $T[r..r']$, we have $p + \mu \in \mathsf{LeftMisper}(T, r, r', |T|)$; see Fig. 5.1.

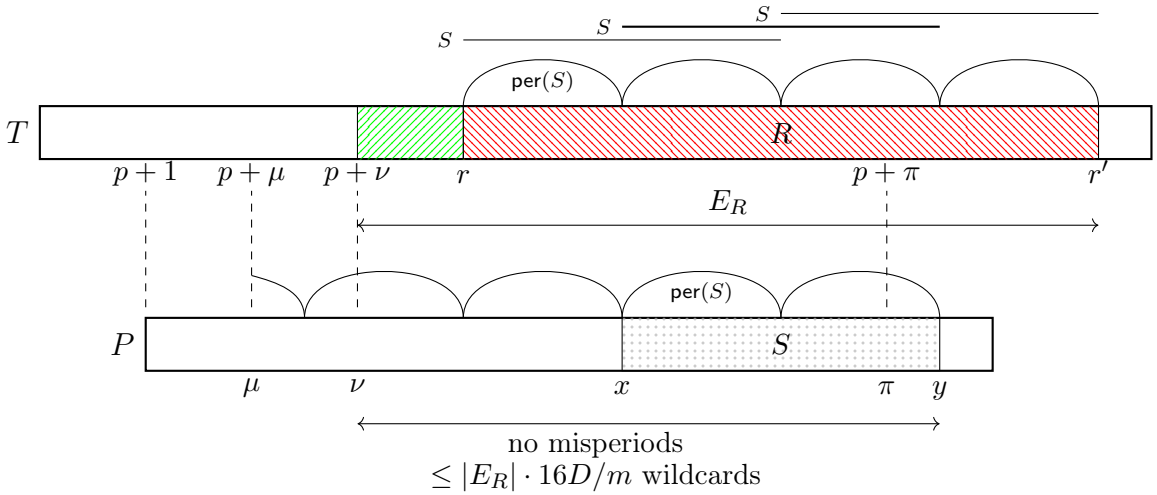

Figure 5.1: The run R and an occurrence p of P in T that aligns S with an occurrence of S in R.

Now, assume for sake of a contradiction that $p + \mu \notin \mathcal{M}_R$. Intuitively, this can only be the case if the extension of R did not reach the beginning of T due to encountering too many misperiods. On the other hand, the fragment $P[\mu..x)S$ of P contains only one misperiod and cannot contain many wildcards, since every position of S is a sparsifier. As a result, a misperiod in T will be aligned with a position of P that is neither a misperiod

nor a wildcard, contradicting the fact that $p+1$ is an occurrence of P in T. Formally, let $p+\nu > p+\mu$ be the misperiod that forced the extension algorithm to stop, i.e. $E_R = T[p+\nu..r']$. (See Fig. 5.1.) Then, the stopping condition implies that $|\mathcal{M}_R| > |E_R| \cdot 20D/m$. All these misperiods belong to the prefix of E_R matching $P[\nu..y]$. On the other hand, due to $\mu < \nu$, we have $[\nu..y] \cap \mathsf{LeftMisper}(P, x, y, |P|) = \emptyset$. Furthermore, every position of S is a sparsifier, and therefore the number of wildcards in $P[\nu..y]$ is at most $|E_R| \cdot 16D/m$. Thus, there exists $p+s \in \mathcal{M}_R$ such that $s \notin \mathsf{LeftMisper}(P, x, y, |P|)$ and $P[s]$ is not a wildcard. This implies that $T[p+s] \neq P[s]$, a contradiction to the fact that $p+1$ is an occurrence of P. □

▷ Claim 5.3.12. The set $\cup_{R\in\mathcal{S}}\mathcal{M}_R$ is of size $O(D)$ and it can be computed using $O(D)$ PILLAR operations given the set § of S-runs and q.

Proof. We start by initialising a set $\mathcal{R} := §$, marking every element of $\mathcal{R}$ as unprocessed and a set $\mathcal{M} = \emptyset$. We then iteratively perform the following procedure for the rightmost unprocessed $R = T[r..r'] \in \mathcal{R}$. Compute $\mathcal{M}_R$ using Lemma 5.3.3, set $\mathcal{M} := \mathcal{M} \cup \mathcal{M}_R$, and mark R as processed. This takes time proportional to the sum of $|\mathcal{M}_R|$ and the number of groups of wildcards contained in E_R. Let us say that two elements $R = T[r..r']$ and $R' = T[t..t']$ of $\mathcal{S}$ are *synchronised* if and only if $r = t \pmod q$. During the procedure, whenever we compute some $E_R = T[x..r']$ that extends beyond an (unprocessed) S-run $R' = T[t..t']$, that is, $x \le t \le t' < r'$, and R and R' are synchronised, we remove R' from $\mathcal{R}$—the total time required for this step is $O(G)$.

We now show the correctness of the algorithm. If, while extending a run $R = T[r..r'] \in \mathcal{R}$, we extend beyond a run $R' = T[t..t'] \in \mathcal{R}$ with $r = t \pmod q$, observe that the left endpoint of E_R cannot be to the right of the left endpoint of $E_{R'}$, since we have at least as big a budget for misperiods in the extension of R when we reach position t as in the extension of R' when we reach position t. This implies that $\mathcal{M}_{R'} \subseteq \mathcal{M}_R$ and hence the algorithm correctly computes $\mathcal{M} = \cup_{R\in\mathcal{S}}\mathcal{M}_R$. Additionally, it guarantees that any computed E_R and $E_{R'}$ for synchronised S-runs R and R' are disjoint.

Finally, we analyse the algorithm's time complexity. Henceforth, $\mathcal{R}$ denotes set of runs that were processed. Observe that the run extensions take $O(\sum_{R\in\mathcal{R}} |\mathcal{M}_R|)$ time in total in the PILLAR model. As we have

$$\begin{aligned}\sum_{R\in\mathcal{R}} |\mathcal{M}_R| &\le |\mathcal{R}| + \sum_{R\in\mathcal{R}} |E_R| \cdot 20D/m \\ &\le O(G) + 20D/m \cdot \sum_{R\in\mathcal{R}} |E_R|,\end{aligned}$$

proving that $\sum_{R\in\mathcal{R}} |E_R| = O(m)$ directly yields that $\mathcal{M} = O(D)$ and that the algorithm takes $O(D)$ time.

In what follows, we ignore all E_R that are of length at most m/D as their total length is $O(G \cdot m/D) = O(m)$. Let us partition T into a collection $\mathcal{Q} = \{T[1+iq..(i+1)q] : i \in [0..\lfloor n/q \rfloor - 1]\}$ of consecutive fragments of length q, with the last one potentially being shorter and in this case discarded. We say that an element $T[i..j]$ of $\mathcal{Q}$ is *synchronised* with an element $R = T[r..r']$ of $\mathcal{R}$ if no position in $[i..j]$ is a misperiod with respect to $T[r..r']$. For a run $R = T[r..r']$, let $\mathcal{Q}_R = \{T[i..j] \in \mathcal{Q} : r \le i \le j \le r'\}$ consist of all

elements of $\mathcal{Q}$ that are fully contained in E_R and observe that

$$\begin{aligned}|\mathcal{Q}_R| &\geq |E_R|/q - 2\\ &\geq |E_R|/(m/256D) - 2\\ &= |E_R| \cdot 256D/m - 2\\ &\geq |E_R| \cdot 252D/m + 2.\end{aligned}$$

Further, let $\mathcal{Q}_R^s = \{X \in \mathcal{Q}_R : X \text{ is synchronised with } R\}$. As E_R contains at most $|E_R| \cdot 20D/m + 1$ misperiods with respect to $T[r..r']$, we have $|\mathcal{Q}_R^s| \geq |\mathcal{Q}_R|/2$. This means that $|E_R| = O(|\mathcal{Q}_R^s| \cdot q)$. Now, observe that if some element of $\mathcal{Q}$ is synchronised with two elements R and R' of $\mathcal{R}$, then R and R' are themselves synchronised. Since the computed extensions of synchronised runs are pairwise disjoint, the considered sets $\mathcal{Q}_R^s$ are pairwise disjoint and hence the bound follows:

$$\begin{aligned}\sum_{R\in\mathcal{R}} |E_R| &= O(m) + \sum_{R\in\mathcal{R}, |E_R|\geq m/D} |E_R|\\ &= O(m + |\mathcal{Q}| \cdot q)\\ &= O(m)\end{aligned}$$

□

We can now conclude the proof of the theorem. By the penultimate claim, the starting positions of occurrences of P in T are in the set $\{\nu - \mu + 1 : \nu \in \mathcal{M}\}$. This concludes the proof of the combinatorial bound, as the size of this set is $O(D)$. As for the time complexity, we verify each candidate position using Fact 5.2.2 in total time $O(DG)$ in the PILLAR model. □

5.4 Pattern Matching with k Mismatches in the PILLAR Model

In this section, we extend the results of Section 5.3, showing that the *k-mismatch occurrences* of a pattern $P \in \Sigma_\diamond^*$ in a solid text T can be computed in $\tilde{O}((D+G)\cdot(G+k))$ time in the PILLAR model. Further, we prove that the starting positions of these occurrences can be decomposed into $O((D+k)G)$ arithmetic progressions with the same difference, plus $O((D+k)k)$ additional k-mismatch occurrences.

Theorem 5.1.2. *Let P be a pattern of length m with D wildcards arranged in G groups, T be a solid text of length $n \leq 3m/2$, and k be a positive integer. We can compute a representation of the k-mismatch occurrences of P in T as $O((D+k)G)$ arithmetic progressions with common difference and $O((D+k)k)$ additional occurrences using $O((D+k)\cdot(G+k)\log\log(D+k))$ time plus $O((D+k)\cdot(G+k))$ PILLAR operations.*

5.4.1 Computing Structure in the Pattern

We start by showing a *decomposition lemma*, that either extracts useful structure from the pattern or reveals that it is close to a periodic string. Our lemma is analogous to the decomposition lemma for the case when both strings are solid [89, Lemma 3.6]. The crucial differences are two:

- in Case (I), we require that breaks are solid strings,
- in Case (II), we ensure that each computed repetitive region contains a sparsifier.

Lemma 5.4.1. *Let P be a string of length m that contains $D \leq m/16$ wildcards arranged in G groups. Further, let $k \in [1..m]$ be an integer threshold, and let $\gamma := G + k$ and $\tau := D + k$. At least one of the following holds:*

(I) P contains 2γ disjoint solid strings $B_1, \ldots, B_{2\gamma}$, that we call breaks*, each having length $m/(16\gamma)$ and the period greater than $m/(512\tau)$.*

(II) P contains r disjoint repetitive regions *$R_1, \ldots, R_r$ of total length $m_R \geq m/8$, such that, for every i:*
- *R_i contains a sparsifier,*
- *$|R_i| \geq m/16\gamma$, and,*
- *for a primitive string Q_i with $|Q_i| \leq m/(512\tau)$, we have $\delta_H(R_i, Q_i^\infty) = \lceil 32k/m \cdot |R_i| \rceil$.*

(III) There exists a primitive string Q of length at most $m/(512\tau)$ such that $\delta_H(P, Q^\infty) \leq 32k$.

Moreover, there is an algorithm that takes $O(G+k)$ time in the PILLAR model and distinguishes between the above cases, returning one of the following: either 2γ disjoint breaks, or repetitive regions $R_1, \ldots, R_r$ of total length at least $m/8$ along with primitive strings $Q_1, \ldots, Q_r$, or a primitive string Q along with $\mathsf{MI}(P, Q)$.

Proof. We process P from left to right. While we have not yet arrived to one of the Cases (I)-(III), we repeat the following procedure. We take the leftmost fragment F of P of length $m/(16\gamma)$ that starts to the right of the current position j and consists only of sparsfiers. If the period of F is greater than $m/(512\tau)$, then we add F to the set of breaks and proceed to position $j + |F|$. Otherwise, we extend F to the right until the either number of mismatches between F and Q^∞, where $Q := F[1..\mathsf{per}(F)]$, becomes equal to $\lceil 32k/m \cdot |F| \rceil$ or we reach the end of P. In the former case, we add F to the set of repetitive regions and proceed to position $j + |F|$. In the latter case, we extend F to the left until either the number of mismatches between F and Q'^∞, where $Q' = \mathsf{rot}^{|F|-m+j}(Q)$, becomes equal to $\lceil 32k/m \cdot |F| \rceil$, or we reach the start of P. If we accumulate enough mismatches, we update the set of repetitive regions to be $\{F\}$ and terminate the algorithm. Otherwise, i.e., if we reach the start of P, we conclude that P is at Hamming distance at most $32k$ from a solid string with period at most $m/(512\tau)$, and we are hence in Case (III).

Let us now prove the correctness of the above algorithm. We first show that if the algorithm has not already concluded that we are in one of the three cases, then there exists a fragment F of sparsifiers that starts in $[j..7m/8]$, where j is our current position. A direct application of Corollary 5.2.5 implies that there are at least $m/2 - D - m/8 \geq 5m/16$ sparsifiers in $[1..7m/8]$ (since $D \leq m/16$), and they are arranged in $G+1$ intervals. By construction, the sparsifiers in an interval are covered from left to right, and hence at most $m/16\gamma - 1$ positions can be left uncovered in each interval. Under our assumptions, the breaks and repetitive regions that have been already computed cover less than $2\gamma \cdot m/(16\gamma) + m/8 = m/4$ positions. If there is no fragment F of length $m/16\gamma$ that consists only of sparsifiers and starts in $[j..7m/8]$, then at least

$$\begin{aligned} 5m/16 - (m/16\gamma - 1) \cdot (G+1) &> 5m/16 - (m/16\gamma) \cdot (G+1) \\ &\geq m/4 \end{aligned}$$

of the sparsifiers have been already covered, a contradiction. From the above, it also follows that if a fragment F reaches the end of the pattern during its extension, then

its length becomes at least $m/8$. Then, if such a fragment is extended to the left and accumulates enough mismatches with respect to the periodicity before the start of P is reached, we can define our set of repetitive regions to be $\{F\}$ and terminate the algorithm. This completes the proof of the correctness of the structural result.

Now, note that the presented proof is algorithmic. Corollary 5.2.5 computes the set of sparsifiers, represented as $O(G)$ disjoint intervals, in $O(G)$ time. After this preprocessing, the procedure can retrieve a new fragment F in constant time. In total, it considers $O(G)$ fragments. Computing the period of a solid string takes $O(1)$ time in the PILLAR model [89, 208]. Moreover, in our attempt to accumulate $O(k)$ misperiods in total, we encounter each group of wildcards at most twice: at most once when extending to the right and at most once when extending to the left. All such extensions thus take $O(G+k)$ time in the PILLAR model due to Lemma 5.3.3. □

5.4.2 The Almost Periodic Case

Case (III) is treated quite similarly to Case II(a) of the exact pattern matching algorithm (see Section 5.3).

Lemma 5.4.2. *Let S be a pattern of length m with D wildcards arranged in G groups[2], let T be a solid text of length n, let $k \in [0..m]$, and let $d \geq 2(k+D)$ be a positive integer. If there exists a primitive string Q with $|Q| \leq m/8d$ such that $\delta_H(S, Q^\infty) \leq \min\{d, 32k\}$, then we can compute a representation of $\mathrm{Occ}_k(S,T)$ as $O(d(G+k))$ arithmetic progressions with common difference $|Q|$ in $O(d(G+k)\log\log d \cdot n/m)$ time plus $O(d \cdot n/m)$ PILLAR operations. Moreover, if $\delta_H(S, Q^\infty) \geq 2k$, then $|\mathrm{Occ}_k(S,T)| = O(d \cdot n/m)$.*

Proof. We only consider the case when $n \leq 3m/2$. The result then follows using the "standard trick" presented in the introduction.

We use an event-driven scheme that extends the one used in the almost periodic case of Section 5.3. First, we apply Corollary 5.3.9 to compute a fragment T' of T that contains the same number of k-mismatch occurrences as T, is at Hamming distance $O(d)$ from a prefix of Q^∞, and only has occurrences of P at positions that are equivalent to 1 (mod $|Q|$). This takes $O(d)$ time in the PILLAR model. We then apply Fact 5.2.1 to compute $\mathsf{MI}(T', Q)$ in $O(d)$ time in the PILLAR model. For a position $i \leq |T'| - m + 1$, the distance d_i between $T'[i..i+m)$ and S is given by

$$d_i = |\mathsf{MI}(S,Q)| + |\mathsf{MI}(T'[i..i+m), Q)| - 2\mathsf{Matching}(i) - \mathsf{Aligned}(i) - \mathsf{Hidden}(i),$$

where

- $\mathsf{Matching}(i)$ is the number of positions that are mismatches between S and Q^∞, and Q^∞ and T', but not between S and T, i.e.,

  $$\mathsf{Matching}(i) = |\{j : j \in \mathsf{MI}(S,Q^\infty) \cap \mathsf{MI}(T'[i..i+m), Q^\infty) \wedge S[j] = T'[i+j] \wedge S[j] \neq \Diamond\}|.$$

- $\mathsf{Aligned}(i)$ is the number of positions that are mismatches between S and Q^∞, Q^∞ and T', and S and T (as opposed to $\mathsf{Matching}(i)$), i.e.,

  $$\mathsf{Aligned}(i) = |\{j : j \in \mathsf{MI}(S,Q^\infty) \cap \mathsf{MI}(T'[i..i+m), Q^\infty) \wedge S[j] \neq T'[i+j] \wedge S[j] \neq \Diamond\}|.$$

- $\mathsf{Hidden}(i)$ is the number of positions that are mismatches between T' and Q^∞, and that are aligned with wildcards, i.e.,

  $$\mathsf{Hidden}(i) = |\{j : j \in \mathsf{MI}(T'[i..i+m), Q^\infty) \wedge S[j] = \Diamond\}|.$$

[2] In the final algorithm, S is a fragment of the pattern P, potentially much shorter than the text.

Recall that every $j \in \mathrm{Occ}_k(S, T')$ satisfies $j \equiv 1 \pmod{|Q|}$ (Theorem 5.1.1), hence we only consider the values of d_i as i increases by multiples of $|Q|$. Then, the value d_i only changes when one of the following events occurs: a position in $\mathsf{MI}(T', Q)$ enters or exits the active window $T'[i..i+m)$, starts or stops being aligned with a group of wildcards, or starts or stops being aligned with a position in $\mathsf{MI}(S, Q)$. As $|\mathsf{MI}(T', Q)| = O(d)$, there are G groups of wildcards in S and $|\mathsf{MI}(S, Q)| = O(k)$, there are $O(d(G+k))$ events.

If $d_i \leq k$, then all positions equivalent to 1 (mod $|Q|$) until the subsequent event are k-mismatch occurrences, and form an arithmetic progression with difference $|Q|$. As there are $O(d(G+k))$ events, we obtain the stated bound on the number of arithmetic progressions.

We sort the events by index in $O(d(G+k)\log\log d)$ time [170]. Then, we process them from left to right; processing one event takes constant time. The initial value of d_0 can be computed in time linear in the number of events. Overall, the running time is dominated by the sorting operation. We additionally perform $O(d)$ PILLAR operations to compute T'.

Finally, we consider the case when $|\mathsf{MI}(S, Q)| \geq 2k$. Let us pick an arbitrary $2k$-size set $M \subseteq \mathsf{MI}(S, Q)$. Then, in every k-mismatch occurrence of S in T, at least $2k - k = k$ misperiods in M are aligned with misperiods in $\mathsf{MI}(T', Q)$, as otherwise there would be more than k mismatches. This observation allows us to apply the marking trick in order to bound the number of k-mismatch occurrences of S in T. For every pair $(x, y) \in M \times \mathsf{MI}(T', Q)$, we place a mark at position $y - x + 1$ of T'. As $|\mathsf{MI}(T', Q)| = O(d)$, we place $O(dk)$ marks. Then, since any position $j \in \mathrm{Occ}_k(S, T')$ must have at least k marks, we have $|\mathrm{Occ}_k(S, T)| = O(dk/k) = O(d)$. □

5.4.3 The Remaining Cases

We now show that in each of the Cases (I) and (II) of Lemma 5.4.1, T contains $O(D+k)$ k-mismatch occurrences of P, and we can efficiently compute a set $\mathcal{S}$ of $O(D+k)$ positions that contains all the starting positions of k-mismatch occurrences of P. We then verify each of the candidate positions in $\mathcal{S}$ in $O(G+k)$ time using Fact 5.2.2.

We first handle the case when the pattern contains 2γ disjoint breaks.

Lemma 5.4.3. *In Case (I) of Lemma 5.4.1, a solid text T of length at most $3m/2$ contains $O(D+k)$ k-mismatch occurrences of P. Moreover, we can compute in $O((D+k)(G+k))$ time in the PILLAR model a set $\mathcal{S} \supseteq \mathrm{Occ}_k(P, T)$ of size $O(D+k)$.*

Proof. Let $\{B_i\}$ be the breaks computed by the algorithm of Lemma 5.4.1. For every i, let p_i denote the starting position of B_i in P. For every exact occurrence j of B_i in T, we put a mark at position $j - p_i + 1$ in T. As B_i has period greater than $m/512\tau$, T contains at most 768τ occurrences of B_i, and they can be computed in $O(|T|/|B_i| + \tau) = O(\gamma + \tau) = O(D+k)$ time in the PILLAR model. As there are 2γ breaks, we place at most $768\tau \cdot 2\gamma = 1536\tau\gamma$ marks in total.

Now, in every occurrence of P, at most k of the B_is are not matched exactly and hence at least $2\gamma - k$ of the B_is are matched exactly. Thus, every position $j \in \mathrm{Occ}_k(P, T)$ has at least $2\gamma - k \geq \gamma$ marks. We designate $\mathcal{S}$ to be the set of positions with at least $2\gamma - k$ marks. Observe that $\mathrm{Occ}_k(P, T) \subseteq \mathcal{S}$ and

$$|\mathcal{S}| \leq 1536\tau\gamma/(2\gamma - k) \leq 1536\tau = O(D+k).$$

Thus, $\mathcal{S}$ satisfies the conditions of the lemma's statement. □

We obtain a similar result for repetitive regions via a more sophisticated marking scheme.

Lemma 5.4.4. *In Case (II) of Lemma 5.4.1, given a solid text T of length at most $3m/2$, we can compute in $O((D+k)(G+k)\log\log(D+k))$ time plus $O((D+k)(G+k))$ PILLAR operations a set $\mathcal{S} \supseteq \mathrm{Occ}_k(P,T)$ of size $O(D+k)$.*

Proof. Let $\{R_i\}$ be the repetitive regions output by the algorithm of Lemma 5.4.1. For every i, let D_i denote the number of wildcards in R_i, $d_i = \lceil 32(k+D)/m \cdot |R_i| \rceil$, and $k_i = \lfloor 16k/m \cdot |R_i| \rfloor$. For every i and every k_i-mismatch occurrence of R_i, we put a mark of weight $|R_i|$ at the corresponding starting position for P.

▷ Claim 5.4.5. For every i, there are $O(D+k)$ k_i-mismatch occurrences of R_i in T. Moreover, the total weight of marks is $O((D+k)\cdot m_R)$.

Proof. We apply Lemma 5.4.2 to each repetitive region. First, let us show that the conditions of the lemma are satisfied. Recall that $|R_i| \geq m/(16(G+k))$ and hence $16k/m \cdot |R_i| \geq 1$.

- $d_i \geq 2(k_i + D_i)$: As R_i contains a sparsifier, we have $D_i \leq 16D/m \cdot |R_i|$ and hence the bound follows.
- $|Q_i| \leq |R_i|/8d_i$: We have $d_i \leq 64|R_i|(k+D)/m \iff m \leq 64|R_i|(k+D)/d_i$ and hence
$$|Q_i| \leq m/(512\tau) = m/(512(k+D)) \leq 64|R_i|/(512d_i) = |R_i|/(8d_i).$$
- $\delta_H(R_i, Q_i^\infty) \leq \min\{d_i, 32k_i\}$: This follows from the fact that $\delta_H(R_i, Q_i^\infty) = \lceil 32k/m \cdot |R_i| \rceil$.

Now, since $\delta_H(R_i, Q_i^\infty) \geq 2k_i$, T contains $O(d_i m/|R_i|)$ occurrences of R_i. Hence, the total weight of marks for a given R_i is
$$w_i = O(d_i m/|R_i|) \cdot |R_i| = O((D+k)\cdot |R_i|),$$
and, summing over $i = 1, \ldots, r \leq 2\gamma$, we get that the total weight of marks that we place is $O((D+k)\cdot m_R)$. □

We next lower bound the number of marks placed at a k-mismatch occurrence of P.

▷ Claim 5.4.6. The total weight of marks placed in any position $\ell \in \mathrm{Occ}_k(P,T)$ is at least $m_R - m/16$.

Proof. Consider a k-mismatch occurrence of P at position ℓ of T. For every i, let r_i be the starting position of R_i in P, and let $k_i' = \delta_H(R_i, T[\ell + r_i .. \ell + r_i + |R_i|))$ be the Hamming distance between R_i and the fragment of T with which it is aligned. As $\ell \in \mathrm{Occ}_k(P,T)$ and the R_is are disjoint, we have $\sum_i k_i' \leq k$. Now, let $I := \{i : k_i' \leq k_i\}$. We have

$$\begin{aligned}
\sum_{i \notin I} |R_i| &= \sum_{i \notin I} \frac{16mk}{16mk} \cdot |R_i| \\
&= \frac{m}{16k} \sum_{i \notin I} \frac{16k}{m} \cdot |R_i| \\
&< \frac{m}{16k} \sum_{i \notin I} k_i' \\
&\leq \frac{m}{16k} \sum_{i=1}^{r} k_i' \\
&\leq \frac{m}{16}.
\end{aligned}$$

The weight of the mark placed at position ℓ in T due to $i \in I$ is $|R_i|$, which amounts to a total weight of at least $m_R - \sum_{i \notin I} |R_i| \geq m_R - m/16$. □

Therefore, we can choose $\mathcal{S}$ to contain all positions to which we have placed marks of total weight at least $m_R - m/16$. Dividing the total weight of marks which is $O((D+k) \cdot m_R)$ by $m_R - m/16$ which is at least $m_R/2$ since $m_R \geq m/8$, we obtain $|\mathcal{S}| = O(D+k)$.

Finding the k_i-mismatch occurrences of R_i using Lemma 5.4.2 requires $O(d_i(G_i + k_i) \log\log d_i \cdot m/|R_i|) = O((D+k)(G_i+k_i)\log\log(D+k))$ time plus $O(d_i \cdot m/|R_i|) = O(D+k)$ PILLAR operations. As the R_is are disjoint, the sum of the G_is is $O(G)$. Further, the sum of the k_is is $O(k)$. Thus, summing over all i, computing the occurrences of all $O(G+k)$ R_is takes $O((D+k)(G+k)\log\log(D+k))$ time plus $O((D+k)(G+k))$ PILLAR operations.

Therefore, computing the set $\mathcal{S}$ of possible starting positions of P in T can be done in $O((D+k)(G+k)\log\log(D+k))$ time plus $O((D+k)(G+k))$ PILLAR operations. □

Lemmas 5.4.3 and 5.4.4 are the last pieces needed to prove the algorithmic part of Theorem 5.1.2.

5.4.4 Proof of the algorithmic part of Theorem 5.1.2.

If P contains $D > m/16$ wildcards, then we apply kangaroo jumping (Fact 5.2.2) to check whether each of the $n \leq 3m/2$ positions in T is a k-mismatch occurrence of P. This requires $O(m(G+k)) = O((D+k)(G+k))$ time in the PILLAR model.

Otherwise, we have $D \leq m/16$, and we can run the algorithm of Lemma 5.4.1, which takes $O(G+k)$ time in the PILLAR model. In Cases (I) and (II), we use Lemmas 5.4.3 and 5.4.4, respectively, to compute using $O((D+k)(G+k)\log\log(D+k))$ time plus $O((D+k)(G+k))$ PILLAR operations a set $\mathcal{S}$ of size $O(D+k)$ that contains all k-mismatch occurrences of P. We verify each of these positions using Fact 5.2.2, in total time $O((D+k)(G+k))$ in the PILLAR model. In Case (III) of Lemma 5.4.1, P satisfies the conditions required for S in Lemma 5.4.2, and we can apply this lemma to compute $\mathrm{Occ}_k(P,T)$ using $O((D+k)(G+k)\log\log(D+k))$ time plus $O(D+k)$ PILLAR operations.

In total, the algorithm uses $O((D+k)(G+k)\log\log(D+k))$ time and $O((D+k)(G+k))$ PILLAR operations.

5.4.5 Proof of the Combinatorial Part of Theorem 5.1.2

Finally, we explain how we can refine the analysis of Lemma 5.4.2 to obtain a more precise characterisation of the structure of k-mismatch occurrences.

Lemma 5.4.7. *The k-mismatch occurrences of P in T can be decomposed into $O((D+k)G)$ arithmetic progressions with common difference q and $O((D+k)k)$ additional occurrences.*

Proof. We proceed similarly to the proof of Lemma 5.4.2, with $S = P$. Let T' be the fragment of T computed by Corollary 5.3.9 using $d = \Theta(D+k)$. We have $|\mathsf{MI}(T', Q)| = O(D+k)$. Define $d'_i = |\mathsf{MI}(P,Q)| + |\mathsf{MI}(T'[i..i+m), Q)| - \mathsf{Hidden}(i)$, where $\mathsf{Hidden}(i) = |\{j : j \in \mathsf{MI}(T'[i..i+m), Q) \wedge P[j] = \Diamond\}|$. Note that $\delta_H(T'[i..i+m), P) = d'_i - 2\mathsf{Matching}(i) - \mathsf{Aligned}(i)$, where

$$\mathsf{Matching}(i) = \{j : j \in \mathsf{MI}(P,Q) \cap \mathsf{MI}(T'[i..i+m), Q) \wedge \Diamond \neq P[j] = T'[i+j]\},$$
$$\mathsf{Aligned}(i) = \{j : j \in \mathsf{MI}(P,Q) \cap \mathsf{MI}(T'[i..i+m), Q) \wedge \Diamond \neq P[j] \neq T'[i+j]\}$$

Therefore, $\delta_H(T'[i..i+m), P) \leq d'_i$. The inequality is strict if at least one of $\mathsf{Matching}(i)$ or $\mathsf{Aligned}(i)$ is positive. In particular, every position where $d'_i \leq k$ corresponds to a k-mismatch occurrence of P in T. Using the event-driven scheme, we compute the values d'_i for all $i \equiv 1 \mod |Q|$. The value d'_i only changes when a position in $\mathsf{MI}(T', Q)$ enters or exits the active window $T'[i..i+m)$, or when a position in $\mathsf{MI}(T', Q)$ starts or stops being aligned with a group of wildcards in P. Therefore, the values d'_i change $O(G \cdot (D+k))$ times. As we only consider positions $i \equiv 1 \mod |Q|$, the set of positions where $d'_i \leq k$ forms $O(G \cdot (D+k))$ arithmetic progressions with common difference $|Q|$.

This analysis might have missed the k-mismatch occurrences where at least one of $\mathsf{Matching}(i), \mathsf{Aligned}(i)$ is positive. However, this requires a misperiod of P to be aligned with a misperiod of T'. As there are $O(k)$ of the former and $O(D+k)$ of the latter, the number of such occurrences is $O((D+k)k)$. □

5.5 Fast Algorithms in Various Settings

In this section, we combine the algorithm encapsulated in Theorem 5.1.2 with efficient implementations of the `PILLAR` model in the standard, dynamic, fully compressed, and quantum settings, thus obtaining efficient algorithms for (approximate) pattern matching with wildcards in these settings. For ease of presentation, we use the terms "exact occurrences" and "0-mismatch occurrences" interchangeably.

5.5.1 The Standard Setting

In the standard setting, where a collection of strings of total length N is given explicitly, `PILLAR` operations can be performed in constant time after an $O(N)$-time preprocessing, cf. [89]. We thus obtain the following result, by noticing that the $\log\log(D+k)$ factor in the complexity of Theorem 5.1.2 only comes from sorting subsets of $[1..n]$ of total size $O((D+k)(G+k))$ in the calls to Lemma 5.4.2, which we can instead do naively in $O(n + (D+k)(G+k))$ time as we can batch the computations.

Theorem 5.5.1. *Let P be a pattern of length m with D wildcards arranged in G groups, T be a solid text of length n, and $k \geq 0$ be an integer. We can compute a representation of the k-mismatch occurrences of P in T in $O(n + (D+k) \cdot (G+k))$ time.*

Remark 5.5.2. We can in fact do better if the size of the alphabet Σ is small in the so-called *packed setting*. Specifically, we can replace the n factor in the complexity of Theorem 5.5.1 by $n/\log_{|\Sigma|} n$ if the input string is given in its packed representation, with each machine word representing $O(\log_{|\Sigma|} n)$ characters, using the `PILLAR` model implementation from [194, 208].

5.5.2 The Compressed Setting

For our purposes, a straight-line program (SLP) is a context-free grammar Γ that consists of the set $\Sigma_\Diamond$ of terminals and a set $N_\Gamma = \{A_1, \ldots, A_n\}$ of non-terminals such that each $A_i \in N_\Gamma$ is associated with a unique production rule $A_i \to f_\Gamma(A_i) \in (\Sigma_\Diamond \cup \{A_j : j < i\})^*$. We can assume without loss of generality that each production rule is of the form $A \to BC$ for some symbols B and C (that is, the given SLP is in Chomsky normal form). Every symbol $A \in S_\Gamma := N_\Gamma \cup \Sigma_\Diamond$ generates a unique string, which we denote by $\mathsf{gen}(A) \in \Sigma_\Diamond^*$.

The string $\mathsf{gen}(A)$ can be obtained from A by repeatedly replacing each non-terminal with its production. We say that Γ generates $\mathsf{gen}(\Gamma) := \mathsf{gen}(A_n)$.

In the fully compressed setting, given a collection of straight-line programs (SLPs) of total size n generating strings of total length N, each `PILLAR` operation can be performed in $O(\log N)$ time after an $O(n \log N)$-time preprocessing, cf. [89, 125]. Additionally, using a dynamic programming approach, we can compute a linked-list representation of all wildcards in P in $O(m+D)$ time. If we applied Theorem 5.1.2 directly in the fully compressed setting, we would obtain $\Omega(N/M)$ time, where N and M are the uncompressed lengths of the text and the pattern, respectively. Instead, we can adapt a dynamic programming approach described in [89, Section 7.2] to obtain the following result.

Theorem 5.5.3 (Fully Compressed Setting)**.** *Let Γ_T denote a straight-line program of size n generating a solid string T, let Γ_P denote a straight-line program of size m generating a string P with D wildcards arranged in G groups, let $k \geq 0$ denote an integer threshold, and set $N := |T|$ and $M := |P|$. We can compute the number of k-mismatch occurrences of P in T in $O(m \log N + n(D+k)(G+k) \log N)$ time. All occurrences can be returned in extra time proportional to the output size.*

5.5.3 The Dynamic Setting

Let $\mathcal{X}$ be a growing collection of non-empty persistent strings; it is initially empty, and then undergoes updates by means of the following operations:

- `Makestring`(U): Insert a non-empty string U into $\mathcal{X}$
- `Concat`(U, V): Insert string UV to $\mathcal{X}$, for $U, V \in \mathcal{X}$
- `Split`(U, i): Insert $U[0..i)$ and $U[i..|U|)$ into $\mathcal{X}$, for $U \in \mathcal{X}$ and $i \in [0..|U|)$.

By N we denote an upper bound on the total length of all strings in $\mathcal{X}$ throughout all updates executed by an algorithm. A collection $\mathcal{X}$ of non-empty persistent strings of total length N can be dynamically maintained with operations `Makestring`(U), `Concat`(U, V), `Split`(U, i) requiring time $O(|U| + \log N)$, $O(\log N)$, and $O(\log N)$, respectively, so that `PILLAR` operations can be performed in time $O(\log N)$; see [125, 160]. All stated time complexities hold with probability $1 - 1/N^{\Omega(1)}$.

It remains to discuss how to compute the linked list representation of the endpoints of groups of wildcards. The data structure of Gawrychowski et al. [160] maintains a *run-length SLP*[3] for each string in the collection, whose height is $O(\log N)$ with probability $1 - 1/N^{\Omega(1)}$. We can straightforwardly compute the sought representation of wildcards in a string $X \in \mathcal{X}$ with D wildcards in $O(D \log N)$ time with probability $1 - 1/N^{\Omega(1)}$, by maintaining at no extra cost, for each non-terminal, whether the string it generates contains a #. We then simply traverse the $O(D \log N)$ nodes of the parse tree that have a descendant terminal symbol that produces # in an in-order fashion, creating groups of wildcards as necessary. The complexity of this step is dominated by the time required by our (approximate) pattern matching algorithm.

Theorem 5.5.4 (Dynamic Setting)**.** *A collection $\mathcal{X}$ of persistent strings of total length N over alphabet $\Sigma_\diamondsuit$ can be dynamically maintained with operations* `Makestring`*,* `Concat` *and* `Split` *requiring time $O(|U| + \log N)$, $O(\log N)$, and $O(\log N)$, respectively, so that, given two strings $P, T \in \mathcal{X}$, such that P has D wildcards arranged in G groups and T is a solid string with $|T| \geq |P|$, and an integer threshold $k \geq 0$, we can return a representation*

[3]The only difference between run-length SLPs and SLPs is that production rules of the form $A \to B^k$ are also allowed.

of all k-mismatch occurrences of P in T in time $O((D+k)(G+k)\cdot|T|/|P|\cdot\log N)$ time. All stated time complexities hold with probability $1-1/N^{\Omega(1)}$.

Kempa and Kociumaka [196, Section 8 in the arXiv version] presented a deterministic implementation of a collection $\mathcal{X}$ of non-empty persistent strings, which allows to remove randomness from the statement of Theorem 5.5.4 at the expense of some multiplicative $\log^{O(1)}\log N$ factors.

5.5.4 The Quantum Setting

In what follows, we assume that the input strings can be accessed in the quantum query model [30, 81]. We are interested in the time complexity of our quantum algorithms [46].

Observation 5.5.5 ([186, Observation 2.3])**.** *For any two strings S,T of length at most n, $\mathsf{LCP}(S,T)$ or $\mathsf{LCP}^R(S,T)$ can be computed in $\tilde{O}(\sqrt{n})$ time in the quantum model with probability at least $1-1/n^c$.*

Fact 5.5.6 (Corollary of [172], cf. [96, Observation 39])**.** *For any strings S and T of length at most n, with $|T|\leq 2|S|$, $\mathsf{IPM}(S,T)$ can be computed in $\tilde{O}(\sqrt{n})$ time in the quantum model with probability at least $1-1/n^c$.*

All other `PILLAR` operations trivially take $O(1)$ quantum time. As a corollary, in the quantum setting, all `PILLAR` operations can be implemented in $\tilde{O}(\sqrt{m})$ quantum time with no preprocessing, as we always deal with strings of length $O(m)$. Additionally, we can compute the linked list representation of the endpoints of groups of wildcards in $\tilde{O}(\sqrt{m}G)$ time in the quantum model with probability at least $1-1/m^c$ as follows: we search for a wildcard and, if we find it, we compute the group that contains it in $O(1)$ time in the `PILLAR` model using Fact 5.2.1; we then recurse on both sides of the group, and so on. In total, this procedure requires $O(G)$ time in the `PILLAR` model and hence $\tilde{O}(\sqrt{m}G)$ time in the quantum model. As a corollary, we obtain the following result.

Theorem 5.5.7 (Quantum Setting)**.** *Consider a string P of length m with D wildcards arranged in G groups, a solid string T of length $n\geq m$, and an integer threshold $k\geq 0$. The k-mismatch occurrences of P in T can be computed in $\tilde{O}((n/\sqrt{m})(G+k)(D+k))$ time in the quantum model with probability at least $1-1/n^c$.*

5.6 A Lower Bound on the Number of Arithmetic Progressions

In this section we show a lower bound on the number of arithmetic progressions covering the set of k-mismatch occurrences of a pattern in a text.

Theorem 5.6.1. *There exist a pattern P of length $m=\Omega((D+k)^{1+o(1)}(k+1))$ and a text T of length $n\leq 3m/2$ such that the set of k-mismatch occurrences of P in T cannot be covered with less than $\Omega((D+k)\cdot(k+1))$ arithmetic progressions.*

Proof. We call a set $S\subseteq[1..n]$ progression-free if it contains no non-trivial arithmetic progression, that is, distinct integers a,b,c such that $a+b-2c=0$.

Fact 5.6.2 ([127])**.** *For any sufficiently large integer M, there exists a progression-free set S of cardinality M that is a subset of $\{1,\ldots,n_M\}$, where $n_M=O(M2^{\sqrt{\log M}})$.*

Let $M = D + k/2$ and let $S \subseteq [n_M]$ be a progression-free set of cardinality M. We encode S as a string P_S of length n_M as follows: for every $i \notin S$ we set $P_S[i] = 0$, and we arbitrarily assign $k/2$ ones and D wildcards to the remaining $D + k/2$ positions. We then consider the pattern $P = 0^\ell P_S 0^\ell$, where ℓ is a parameter to be determined later. In what follows, let $m := 2\ell + n_M$ denote the length of P.

Now, let $M' = k/2 + 1$ and $S' \subseteq [n_{M'}]$ be a progression-free set of cardinality M'. We set

$$T := 0^{m/2} B_1 \dots B_{n_{M'}} 0^{m/2},$$

where $B_i = 0^{t-1}1$ if $i \in S'$, $B_i = 0^t$ otherwise, and $t = \lfloor m/(2n_{M'}) \rfloor$. We pick ℓ large enough such that $t \geq 10n_M$ and $2n_{M'}$ divides m. By construction, T has length $m + t \cdot n_{M'} = 3m/2$. The condition $t \geq 10n_M$ gives the following condition on ℓ:

$$\begin{aligned} t \geq 10n_M &\Leftrightarrow m/2n_{M'} \geq 10n_M \\ &\Leftrightarrow (2\ell + n_M)/2n_{M'} \geq 10n_M \\ &\Leftrightarrow (2\ell + n_M) \geq 20n_M n_{M'} \\ &\Leftrightarrow 2\ell \geq n_M(20n_{M'} - 1) \\ &\Leftrightarrow \ell \geq n_M(10n_{M'} - 1/2) \end{aligned}$$

In particular, the condition $t \geq 10n_M$ holds when $\ell \geq 10n_M n_{M'}$. In this case, we have

$$m = \Omega((k + D)(k + 1)2^{\sqrt{\log(k+D)} + \sqrt{\log(k+1)}}).$$

Let $X = \mathrm{Occ}_k(P, T)$, i.e. it is the set of k-mismatch occurrences of P in T. Observe that $i \in X$ if and only if there exists j such that $P[j] \in \{1, \lozenge\}$ and $T[i + j] = 1$. Moreover, any pair of 1s in T are at least $t \geq 10n_M$ positions apart, while the oness and wildcards of P all lie within an interval of size n_M. Therefore, for a given alignment of P and T, there can be at most one 1 of T that is aligned with a 1 or a wildcard of P; it follows that $\mathrm{Occ}_k(P, T)$ has cardinality

$$(D + k/2) \cdot (k/2 + 1) = \Omega((D + k) \cdot (k + 1)).$$

It remains to show that $X =$ does not contain arithmetic progressions of length 3. Assume for a sake of contradiction that there exist $x, y, z \in X$ with $x < y < z$ that form an arithmetic progression, i.e.,

$$y - x = z - y.$$

Let i_x denote the index of the block B_{i_x} of T that contains the leftmost 1 that is aligned with a 1 or a wildcard of P: this 1 is at position $m/2 + i_x t$ in T. Similarly, let d_x be such that the corresponding aligned character is at position $\ell + d_x$ in P. Note that, by construction, i_x is an element of S' and d_x is an element of S, both of which are progression-free sets. Define i_y, i_z, d_y, d_z similarly for y, z. To obtain the sought contradiction, we show that one of (i_x, i_y, i_z) or (d_x, d_y, d_z) is an arithmetic progression. We can express each $w \in \{x, y, z\}$ in terms of i_w and d_w as

$$w = m/2 + i_w t - d_w - \ell + 1.$$

Combining the above equations for x, y, and z, we get

$$y - x = (i_y - i_x)t - (d_y - d_x) \text{ and } z - y = (i_z - i_y)t - (d_z - d_y).$$

By construction, $|d_y - d_x| \leq n_M$. As $t \geq 10n_M$, the equality $y - x = z - y$ thus yields

$$i_y - i_x = i_z - i_y \text{ and } d_y - d_x = d_z - d_y.$$

However, as $x < y < z$, at least one of the above two equations involves non-zero values. In other words, there is a three-term arithmetic progression in either S or S', contradicting the fact that they are progression-free. □

Chapter 6

Longest Common Extension with Wildcards and applications: Pattern Matching and Matrix Multiplication

6.1 Introduction

Given a string T, the *longest common extension* (LCE) at indices i and j is the length of the longest common prefix of the suffixes of T starting at indices i and j. In the LCE problem, given a string T, the goal is to build a data structure that can efficiently answer LCE queries.

Longest common extension queries are a powerful string operation that underlies a myriad of string algorithms, for problems such as approximate pattern matching [20, 32, 89, 147, 224, 225], finding maximal or gapped palindromes [64, 95, 168, 211], and computing the repetitive structure (e.g., runs) in strings [45, 209], to name just a few.

Due to its importance, the LCE problem and its variants have received a lot of attention [64, 66, 67, 68, 69, 138, 157, 160, 171, 194, 197, 198, 202, 215, 246, 252, 276, 277, 278]. The suffix tree of a string of length n occupies $\Theta(n)$ space and can be preprocessed in $O(n)$ time to answer LCE queries in constant time [138, 171]. However, the $\Theta(n)$ space requirement can be prohibitive for applications such as computational biology that deal with extremely large strings. Consequently, much of the recent research has focused on designing data structures that use less space without being (much) slower in answering queries. Consider the setting when we are given a read-only length-n string T over an alphabet of size polynomial in n. Bille et al. [66] gave a data structure for the LCE problem that, for any given user-defined parameter $\tau \leq n$, occupies $O(\tau)$ space on top of the input string and answers queries in $O(n/\tau)$ time. Kosolobov [214] showed that this data structure is optimal when $\tau = \Omega(n/\log n)$. A drawback of the data structure of Bille et al. [66] is its rather slow $O(n^{2+\varepsilon})$ construction time. This motivated studies towards an LCE data structure with optimal space and query time and a fast construction algorithm. Gawrychowski and Kociumaka [157] gave an optimal $O(n)$-time and $O(\tau)$-space Monte Carlo construction algorithm and Birenzwige et al. [69] gave a Las Vegas construction algorithm with the same complexity provided $\tau = \Omega(\log n)$. Finally, Kosolobov and Sivukhin [215] gave a deterministic construction algorithm that works in optimal $O(n)$ time and $O(\tau)$ space for $\tau = \Omega(n^{\varepsilon})$, where $\varepsilon > 0$ is an arbitrary constant. Another line of work [67, 68, 160, 197, 198, 246, 276, 277, 278] considers LCE data structures over compressed strings.

One important variant of the LCE problem is that of LCE with k-mismatches (k-LCE), where one wants to find the longest prefixes that differ in at most k positions, for a given integer parameter k. Landau and Vishkin [224] proposed a technique, dubbed "kangaroo jumping", that reduces k-LCE to $k+1$ standard LCE queries. This technique is a central component of many approximate pattern matching algorithms, under the Hamming [32, 89] and the edit [20, 89, 224] distances.

In this work, we focus on the variant of LCE in strings with *wildcards*, denoted LCEW. Wildcards (also known as *holes* or *don't cares*), denoted $\lozenge$, are special characters that match every character of the alphabet. Wildcards are a versatile tool for modeling uncertain data, and algorithms on strings with wildcards have garnered considerable attention in the literature [10, 15, 20, 32, 52, 65, 71, 72, 74, 100, 102, 103, 106, 111, 113, 140, 141, 163, 180, 184, 189, 236, 244, 245, 253].

Given a string T, and indices i, j, $\mathsf{LCEW}(i, j)$ is the length of the longest matching prefixes of the suffixes of T starting at indices i and j. For all $\tau \in [1..n]$, Iliopoulos and Radoszewski [180] showed an LCEW data structure with $O(n^2 \log n/\tau)$ preprocessing time, $O(n^2/\tau)$ space, and $O(\tau)$ query time. In the case where the number of wildcards in T is bounded, more efficient data structures exist.

The LCEW problem is closely related to k-LCE: if we let $\widehat{T}$ be the string obtained by replacing each wildcard in T with a new character, the i-th wildcard replaced with a fresh letter $\#_i$, then an LCEW query in T can be reduced to a D-LCE query in $\widehat{T}$, where D is the number of wildcards in T. Consequently, an LCEW query can be answered using $O(D)$ LCE queries. In particular, if we use the suffix tree to answer LCE queries, we obtain a data structure with $O(n)$ space and construction time and $O(D)$ query time. At the other end of the spectrum, Blanchet-Sadri and Lazarow [70] showed that one can achieve $O(1)$ query time using $O(nD)$ space after an $O(nD)$-time preprocessing.

By using the structure of the wildcards inside the string, one can improve the aforementioned bounds even further. Namely, it is not hard to see that if the wildcards in T are arranged in G maximal contiguous groups (see Example 6.1.1), then we can reduce the number of LCE queries needed to answer an LCEW query to G by jumping over such groups, thus obtaining a data structure with $O(n)$-time preprocessing, $O(n)$ space, and $O(G)$ query time. On the other hand, Crochemore et al. [117] devised an $O(nG)$-space data structure that can be built in $O(nG)$ time and can answer LCEW queries in constant time.

Example 6.1.1. In the string $T = \texttt{abab}\lozenge\lozenge\lozenge\texttt{aaaa}\lozenge\lozenge\lozenge\lozenge\texttt{ba}\lozenge\lozenge\lozenge\texttt{bb}$, we have $D = 10$ and $G = 3$.

6.1.1 Our Results

In this work, we present an LCEW data structure that achieves a smooth space-time trade-off between the data structure based on "kangaroo jumps" and that of Crochemore et al. [117]. More precisely, for any $1 \leq t \leq G$, we show that there exists a data structure using $O(nG/t)$ space and answering queries in time $O(t)$ (see Theorem 6.1.2 for more details). As our main contribution, we show that for any $t \leq G$, there exists a set of $O(G/t)$ positions, called *selected* positions, that intersects any chain of t kangaroo jumps from a fixed pair of positions. Given the LCEW information on selected positions, we can speed up LCEW queries on arbitrary positions by jumping from the first selected position in the common extension to the last selected position in the common extension. This gives us an $O(t)$ bound on the number of kangaroo jumps we need to perform to answer a query. We leverage the fast FFT-based algorithm of Clifford and Clifford [100] for pattern

matching with wildcards to efficiently build a dynamic programming table containing the result of LCEW queries on pairs of indices containing a selected position; this table allows us to jump from the first to the last selected position in the common extension in constant time. The size of the table is $O(nG/t)$, while the query time is $O(t)$. For comparison, the data structures of Crochemore et al. [117] and of Iliopoulos and Radoszewski [180] use a similar dynamic programming scheme that precomputes the result of LCEW queries for a subset of positions: Crochemore et al. use all transition positions (see Section 6.3 for a definition), while Iliopoulos and Radoszewski use one in every $\sqrt{n}$ positions. We use a more refined approach, that allows us to obtain both a dependency on G instead of n and a space-query-time trade-off. Our result can be stated formally as follows.

Theorem 6.1.2. *Suppose that we are given a string T of length n that contains wildcards arranged into G maximal contiguous groups. For every $t \in [1..G]$, there exists a deterministic data structure that:*

- *uses space $O(nG/t)$,*
- *can be built in time $O(n(G/t)\log n)$ using $O(nG/t)$ space,*
- *given two indices $i, j \in [1..n]$, returns $\mathsf{LCEW}(i, j)$ in time $O(t)$.*

We further show that this trade-off can be extended to $t \geq G$ by implementing the kangaroo jumping method of Landau and Vishkin [224] with a data structure that provides a time-space trade-off for (classical) LCE queries. Using the main result of Kosolobov and Sivukhin [215], we obtain the following:

Corollary 6.1.3. *Suppose that we are given a read-only string T of length n that contains wildcards arranged into G maximal contiguous groups. For every constant $\varepsilon > 0$ and $t \in [G..G \cdot n^{1-\varepsilon}]$, there exists a data structure that:*

- *uses space $O(nG/t)$,*
- *can be built in time $O(n)$ using $O(nG/t)$ space,*
- *given two indices $i, j \in [1..n]$, returns $\mathsf{LCEW}(i, j)$ in time $O(t)$.*

Proof. We build the LCE data structure of Kosolobov and Sivukhin [215] for parameter $\tau = nG/t = \Omega(n^{\varepsilon})$ in $O(n)$ time and $O(\tau)$ space. As an LCEW query reduces to G LCE queries, the constructed data structure supports LCEW queries in $O(G \cdot n/\tau) = O(t)$ time. □

By a reduction from Boolean matrix multiplication (BMM), we derive a conditional $\Omega(n^{2-o(1)})$ lower bound on the product of the preprocessing and query times of any combinatorial data structure for the LCEW problem (Theorem 6.4.2).[1] This is the first lower bound for this problem and matches the trade-off of our data structure up to subpolynomial factors when $G = \Theta(n)$.

Surprisingly, one can also use the connection between the two problems to derive an algorithm for sparse Boolean matrix multiplication. Existing algorithms for BMM can be largely categorised into two types: combinatorial and those relying on (dense) fast matrix multiplication. However, the latter are notorious for the significant hidden constants in their asymptotic complexity, making them unlikely candidates for practical applicability. By using the connection to the LCEW problem, we show a deterministic combinatorial algorithm with runtime $\tilde{O}(n\sqrt{m_{in} \cdot (n + m_{out})})$. Our algorithm ties or outperforms all other known deterministic combinatorial algorithms [169, 220, 221, 283] for some range

[1] In line with previous work, we say that an algorithm or a data structure is *combinatorial* if it does not use fast matrix multiplication as a subroutine during preprocessing or while answering queries.

Table 6.1: Overview of combinatorial deterministic sparse Boolean matrix multiplication algorithms. The values m_{in} (resp. m_{out}) refer to the total number of non-zero entries in the input matrices (resp., in the output matrix).

Source	Running Time
Gustavson [169]	$O(n \cdot m_{in})$
Kutzkov [221]	$O(n \cdot (n + m_{out}^2))$
Künnemann [220]	$O(\sqrt{m_{out}} \cdot n^2 + m_{out}^2)$
Abboud et al. [12]	$\tilde{O}(m_{in}\sqrt{m_{out}})$
Our algorithm	$\tilde{O}(n\sqrt{m_{in} \cdot (n + m_{out})})$

of parameters m_{in} and m_{out}, e.g., when $m_{in} = \Theta(n^{3/2})$ and $m_{out} = \Theta(n^{4/3})$, except for the one implicitly implied by the result of Abboud et al. [12]. Namely, by replacing fast matrix multiplication (used in a black-box way) in [12, Theorem 4.1] with the naive matrix multiplication algorithm, one obtains a deterministic combinatorial algorithm with runtime $\tilde{O}(m_{in}\sqrt{m_{out}})$, which is always better than our time bound. See Table 6.1 for a summary. However, our algorithm is much simpler than that of Abboud et al. [12]: while our algorithm relies solely on standard tools typically covered in undergraduate computer science courses, theirs requires an intricate construction of a family of hash functions with subsequent derandomisation. We provide a (non-optimized) proof-of-concept implementation at https://github.com/GBathie/LCEW.

Applications

We further showcase the significance of our data structure by using it to improve over the state-of-the-art algorithms for approximate pattern matching and the construction of periodicity-related arrays for strings containing wildcards.

As previously mentioned, LCE queries play a crucial role in string algorithms, especially in approximate pattern matching algorithms, such as for the problem of *pattern matching with k errors* (k-PME, also known as pattern matching with k *edits* or *differences*). This problem involves identifying all positions in a given text where a fragment starting at that position is within an edit distance of k from a given pattern. The now-classical algorithm of Landau and Vishkin [224] elegantly solves this problem, achieving a time complexity of $O(nk)$ through extensive use of LCE queries. A natural extension of k-PME is the problem of *pattern matching with wildcards and k-errors* (k-PMWE), where the pattern and the text may contain wildcards. The algorithm of Landau and Vishkin [224] for pattern matching under the edit distance can be extended to an $O(n(k + G))$-time algorithm for k-PMWE in strings with G groups of wildcards (see [20]). Building on their work, Akutsu [20] gave an algorithm for k-PMWE that runs in time $\tilde{O}(n\sqrt{km})$. In Theorem 6.5.2, we give an algorithm for k-PMWE with runtime $O(n(k + \sqrt{Gk \log n}))$, which improves on the algorithms of Akutsu [20] and Landau and Vishkin [224] in the regime where $k \ll G \ll m$.

Periodicity arrays capture repetitions in strings and are widely used in pattern matching algorithms, see e.g. [114, 115]. The prefix array of a length-n string T with wildcards

stores $\mathsf{LCEW}(1, j)$ for all $1 \leq j \leq n$. It was first studied in [181], where an $O(n^2)$-time construction algorithm was given. More recently, Iliopoulos and Radoszewski [180] presented an $O(n\sqrt{n \log n})$-time and $\Theta(n)$-space algorithm. Another fundamental periodicity array is the border array, which stores the maximum length of a proper border of each prefix of the string. When a string contains wildcards, borders can be defined in two different ways [175, 181]. A *quantum border* of a string T is a prefix of T that matches the same-length suffix of T, while a *deterministic border* is a border of a string T' that does not contain wildcards and matches T, see Example 6.1.4. A closely related notion is that of quantum and deterministic periods and their respective period arrays (see Section 6.2 for definitions).

Example 6.1.4. The maximal length of a quantum border of $T = \texttt{ab}\lozenge\texttt{bc}$ is 3; note that $\texttt{ab}\lozenge$ matches $\lozenge\texttt{bc}$. The maximal length of a deterministic border of T, however, is 0.

Early work in this area [175, 181] showed that both variants of the border array can be constructed in $O(n^2)$ time. Iliopoulos and Radoszewski [180] demonstrated that one can compute the border arrays from the prefix array in $O(n)$ time and $O(n)$ space, and the period arrays in $O(n \log n)$ time and $O(n)$ space, thus deriving an $O(n\sqrt{n \log n})$-time, $O(n)$-space construction algorithm for all four arrays. In Theorem 6.5.3, we give $O(n\sqrt{G} \log n)$-time, $O(n)$-space algorithms for computing the prefix array, as well as the quantum and deterministic border and period arrays, improving all previously known algorithms when $G = o(n/\log n)$.

6.2 Preliminaries

A string S of length $n = |S|$ is a finite sequence of n characters over a finite alphabet Σ. The i-th character of S is denoted by $S[i]$, for $1 \leq i \leq n$, and we use $S[i..j]$ to denote the *fragment* $S[i]S[i+1]\ldots S[j]$ of S (if $i > j$, then $S[i..j]$ is the empty string). Moreover, we use $S[i..j)$ to denote the fragment $S[i..j-1]$ of S. A fragment $S[i..j]$ is a *prefix* of S if $i = 1$ and a *suffix* of S if $j = n$.

In this chapter, the alphabet Σ contains a special character $\lozenge$ that matches every character in the alphabet. Formally, we define the "match" relation, denoted $\sim$ and defined over $\Sigma \times \Sigma$ as follows:

$$\forall a, b \in \Sigma : a \sim b \Leftrightarrow a = b \vee a = \lozenge \vee b = \lozenge.$$

Its negation is denoted $a \nsim b$. We extend this relation homomorphically to strings of equal length n by $X \sim Y \Leftrightarrow \forall i = 1, \ldots, n : X[i] \sim Y[i]$.

Longest common extensions. Let T be a string of length n, and let $i, j \leq n$ be indices. The longest common extension at i and j in T, denoted $\mathsf{LCE}_T(i, j)$ is defined as the maximum length ℓ such that substrings of T of length ℓ starting at position i and j are equal. Formally:

$$\mathsf{LCE}_T(i, j) = \max\{\ell \leq \min(n - i, n - j) + 1 : T[i..i + \ell) = T[j..j + \ell)\}.$$

Similarly, the longest common extension *with wildcards*, denoted $\mathsf{LCEW}_T(i, j)$, is defined using the $\sim$ relation instead of equality:

$$\mathsf{LCEW}_T(i, j) = \max\{\ell \leq \min(n - i, n - j) + 1 : T[i..i + \ell) \sim T[j..j + \ell)\}.$$

We focus on data structures for LCEW queries inside a string T, but our results can easily be extended to answer queries between two strings P, Q, denoted $\mathsf{LCEW}_{P,Q}(i,j)$. If we consider $T = P \cdot Q$, then for any $i \leq |P|$ and $j \leq |Q|$, we have $\mathsf{LCEW}_{P,Q}(i,j) = \min(\mathsf{LCEW}_T(i, j+|P|), |P|-i+1, |Q|-j+1)$. When the string(s) that we query are clear from the context, we drop the T or P, Q subscripts.

Periodicity arrays. The *prefix array* of a string S of length n is an array π of size n such that $\pi[i] = \mathsf{LCEW}(1, i)$.

An integer $b \in [1..n]$ is a *quantum border* of S if $S[1..b] \sim S[n-b+1..n]$. It is a *deterministic border* of S if there exists a string X *without wildcards* such that $X \sim S$ and $X[1..b] = X[n-b+1..n]$. Similarly, an integer $p \leq n$ is a *quantum period* of S if for every $i \leq n-p, S[i] \sim S[i+p]$, and it is a *deterministic period* of S if there exists a string X *without wildcards* such that $X \sim S$ and for every $i \leq n-p, X[i] = X[i+p]$.

Example 6.2.1. Consider string `ab◊b◊bcb`. Its smallest quantum period is 2, while its smallest deterministic period is 4.

For a string S of length n, we define the following arrays of length n:
- the period array π, where $\pi[i] = \mathsf{LCEW}(1, i)$;
- the deterministic and quantum border arrays, B and B_Q, where $B[i]$ and $B_Q[i]$ are the largest deterministic and quantum border of $S[1..i]$, respectively;
- the deterministic and quantum period arrays, P and P_Q, such that $P[i]$ and $P_Q[i]$ are the smallest deterministic and quantum periods of $S[1..i]$, respectively.

Fact 6.2.2 ([180, Lemmas 12 and 15])**.** *Given the prefix array of a string S, one can compute the quantum border array and quantum period array in $O(n)$ time and space, while the deterministic border and period arrays can be computed in $O(n \log n)$ time and $O(n)$ space.*

6.3 Time-Space Trade-off for LCEW

In this section, we prove Theorem 6.1.2, which we recall below.

Theorem 6.1.2. *Suppose that we are given a string T of length n that contains wildcards arranged into G maximal contiguous groups. For every $t \in [1..G]$, there exists a deterministic data structure that:*
- *uses space $O(nG/t)$,*
- *can be built in time $O(n(G/t)\log n)$ using $O(nG/t)$ space,*
- *given two indices $i, j \in [1..n]$, returns $\mathsf{LCEW}(i,j)$ in time $O(t)$.*

Following the work of Crochemore et al. [117], we define *transition positions* in T, which are the positions at which T transitions from a block of wildcards to a block of non-wildcards characters. We use Tr to denote the set of transition positions in T. Formally, a position $i \in [1..n]$ is in Tr if one of the following holds:
- $i = n$, or
- $i > 1$, $T[i-1] = \lozenge$ and $T[i] \neq \lozenge$.

Note that as T contains G groups of wildcards, there are at most $G+1$ transition positions, i.e., $|\mathsf{Tr}| = O(G)$. Moreover, by definition, the only transition position i for which $T[i]$ may be a wildcard is n.

Our algorithm precomputes the LCEW information for a subset of evenly distributed transition positions, called *selected positions* and denoted Sel, whose number depends on the parameter t. The set Sel contains one in every t transition position in Tr, along with the last one (which is n). Formally, let $i_1 < i_2 < \ldots < i_r$ denote the transition positions of T, sorted in increasing order, then $\mathsf{Sel} = \{i_{st+1} : s = 0, \ldots, \lfloor (r-1)/t \rfloor\} \cup \{n\}$. Let λ denote the cardinality of Sel, which is $O(G/t)$.

Additionally, for every $i \in [1..n]$, we define $\mathsf{next\_tr}[i]$ (resp., $\mathsf{next\_sel}[i]$) as the distance between i and the next transition position (resp., the next selected position) in T. Formally, $\mathsf{next\_tr}[i] = \min\{j - i : j \in \mathsf{Tr} \wedge j \geq i\}$ and $\mathsf{next\_sel}[i] = \min\{j - i : j \in \mathsf{Sel} \wedge j \geq i\}$. These values are well-defined: as n is both a transition and a selected position, the minimum in the above equations is never taken over the empty set. Both arrays can be computed in linear time and stored using $O(n)$ space. The array next_tr can be used to jump from a wildcard to the end of the group of wildcards containing it, due the following property:

Observation 6.3.1. *For any i such that $T[i] = \Diamond$, let $r = \mathsf{next\_tr}[i]$. We have $T[i..i+r) = \Diamond^r$, i.e., the fragment from i until the next transition position (exclusive) contains only wildcards.*

The central component of our data structure is a dynamic programming table, Jump, which allows us to efficiently answer LCEW queries for selected positions. For each selected position i and each (arbitrary) position j, this table stores the distance from i to the last selected position that appears in the common extension on the side of i, i.e. the last selected position i' for which $T[i..i']$ matches $T[j..j+i'-i]$. More formally:

$$\forall i \in \mathsf{Sel}, j \in [1..n] : \textsc{Jump}[i,j] = \max\{i' - i : i' \in \mathsf{Sel} \wedge i' \geq i \wedge T[i..i'] \sim T[j..j+i'-i]\}.$$

If there is no such selected position i' (which happens when $T[i] \nsim T[j]$), then we let $\textsc{Jump}[i,j] = -\infty$. This table contains $\lambda \cdot n = O(nG/t)$ entries and allows us to jump from the first to the last selected position in the common extension, thus reducing LCEW queries to finding longest common extensions *to* and *from* a selected position.

Finally, let $T_\#$ be the string obtained by replacing all wildcards in T with a new character "#" that does not appear in T. The string $T_\#$ does not contain wildcards, and for any $i, j \in [1..n]$, we have $\mathsf{LCEW}_T(i,j) \geq \mathsf{LCE}_{T_\#}(i,j)$.

The data structure. Our data structure consists of

- the Jump table,
- the arrays next_tr and next_sel, and
- a data structure for constant-time LCE queries in $T_\#$, with $O(n)$ construction time and $O(n)$ space usage (e.g. a suffix array or a suffix tree [138]).

The Jump table uses space $O(nG/t)$ and can be computed in time $O(n(G/t)\log n)$ (see Section 6.3.1), while the next_tr and next_sel arrays can be computed in $O(n)$ time and stored using $O(n)$ space. Therefore, our data structure can be built in $O(n + n(G/t) \cdot \log n) = O(n(G/t) \cdot \log n)$ time and requires $O(n + nG/t) = O(nG/t)$ space. As shown in Section 6.3.2, we can use this data structure to answer LCEW queries in T in time $O(t)$, thus proving Theorem 6.1.2.

6.3.1 Computing the Jump Table

In this section, we prove the following lemma.

Lemma 6.3.2. *Given random access to T, the* Jump *table can be computed in $O(n(G/t)\cdot \log n)$ time and $O(nG/t)$ space.*

Proof. To compute the Jump table, we leverage the algorithm of Clifford and Clifford [100] for exact pattern matching with wildcards. This algorithm runs in time $O(n \log m)$, and finds all occurrences of a pattern of length m within a text of length n (both may contain wildcards).

Let $i_1 < i_2 < \ldots < i_\lambda = n$ denote the *selected* positions, sorted in increasing order. For $r = 1, \ldots, \lambda - 1$, let P_r be the fragment of T from the r-th to the $(r+1)$-th selected position (exclusive), i.e., $P_r = T[i_r..i_{r+1})$, and let ℓ_r denote the length of P_r, that is $\ell_r = |P_r| = i_{r+1} - i_r$. Then, for every r, we use the aforementioned algorithm of Clifford and Clifford [100] to compute the occurrences of P_r in T: it returns an array A_r such that $A_r[i] = 1$ if and only if $T[i..i+\ell_r) \sim P_r$.

Using the arrays $(A_r)_r$, the Jump table can then be computed with a dynamic programming approach, in the spirit of the computations in [117]. The base case is $i = i_\lambda$, for which we have, for all $j \in [1..n]$, $\textsc{Jump}[i_\lambda, j] = 0$ if $T[i_\lambda] \sim T[j]$ and $-\infty$ otherwise. We can then fill the table by iterating over all pairs $(r, j) \in [1..\lambda - 1] \times [1..n]$ in the reverse lexicographical order and using the following recurrence relation:

$$\textsc{Jump}[i_r, j] = \begin{cases} -\infty & \text{if } T[i_r] \nsim T[j] \\ \max(0, \ell_r + \textsc{Jump}[i_{r+1}, j + \ell_r]) & \text{if } T[i_r] \sim T[j], A_r[j] = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{6.1}$$

Computing the arrays $(A_r)_r$ takes $O(\lambda \cdot n \log n) = O(n(G/t) \cdot \log n)$ time in total. Computing the Jump table from the arrays takes constant time per cell, and the table contains $\lambda \cdot n = O(nG/t)$ cells. Thus, the Jump table can be computed in time $O(n(G/t)\cdot \log n)$. □

6.3.2 Answering LCEW Queries

Our algorithm to answer an LCEW query can be decomposed into the following steps:

(a) move forward in T until we reach a selected position or a mismatch,
(b) use the Jump table to skip to the last selected position in the longest common prefix on the side of the selected position,
(c) move forward until we either reach a mismatch or the end of the text.

Steps (a) and (b) might have to be performed twice, one for each of the "sides" of the query. Steps (a) and (c) can be handled similarly, using LCE queries in $T_\#$ to move forward multiple positions at a time: see Algorithm 1 for the pseudo-code for these steps, and see Algorithm 2 for the pseudo-code of the query procedure.

Analysis of the NextSelectedOrMismatch subroutine (Algorithm 1). Algorithm 1 computes a value ℓ such that $T[i..i+\ell) \sim T[j..j+\ell)$, and either $T[i+\ell] \nsim T[j+\ell]$ or at least one of $i+\ell, j+\ell$ is a selected position. In the latter case, $i+\ell$ (resp. $j+\ell$) is the first selected position after i (resp. j). Furthermore, Algorithm 1 runs in time $O(t)$. These properties are formally stated below.

Lemma 6.3.3. *Let ℓ be the value returned by Algorithm 1. We have $T[i..i+\ell) \sim T[j..j+\ell)$.*

Algorithm 1 Subroutine for LCEW queries

```
1: function NextSelectedOrMismatch(i, j)
2:     ℓ ← 0
3:     m ← min(next_sel[i], next_sel[j])
4:     while T[i + ℓ] ∼ T[j + ℓ] and i + ℓ ∉ Sel and j + ℓ ∉ Sel do
5:         r ← LCE_{T#}(i + ℓ, j + ℓ)
6:         ℓ ← min(ℓ + r, m)
7:         d ← 0
8:         if T[i + ℓ] = ◊ then
9:             d ← max(d, next_tr[i + ℓ])
10:        if T[j + ℓ] = ◊ then
11:            d ← max(d, next_tr[j + ℓ])
12:        ℓ ← min(ℓ + d, m)
13:    return ℓ
```

Proof. We prove that $T[i..i+\ell) \sim T[j..j+\ell)$ by induction on the number of iterations of the `while` loop. At the start of the algorithm, we have $\ell = 0$, $i = i + \ell$ and $j = j + \ell$, hence the base case holds.

Now, assume that at the start of some iteration of the `while` loop, we have $T[i..i+\ell) \sim T[j..j+\ell)$. In Algorithm 1, $r = \mathsf{LCE}_{T_\#}(i+\ell, j+\ell)$, and hence $T_\#[i+\ell..i+\ell+r)$ is equal to $T_\#[j+\ell..j+\ell+r)$, and *a fortiori*, the same thing is true in T, i.e., $T[i+\ell..i+\ell+r) \sim T[j+\ell..j+\ell+r)$. Combining the above with our invariant hypothesis, we obtain that $T[i..i+\ell+r) \sim T[j..j+\ell+r)$. Therefore, after Algorithm 1 is executed, we have $T[i..i+\ell) \sim T[j..j+\ell)$ for the new value of ℓ. By Observation 6.3.1, at least one of $T[i+\ell..i+\ell+d)$ or $T[j+\ell..j+\ell+d)$ consists only of wildcards, therefore these two fragments match, and, before executing Algorithm 1, we have $T[i..i+\ell+d) \sim T[j..j+\ell+d)$. Finally, after executing Algorithm 1, the above becomes $T[i..i+\ell) \sim T[j..j+\ell)$, and our induction hypothesis holds. □

The fact that either $T[i+\ell] \nsim T[j+\ell]$ or at least one of $i+\ell, j+\ell$ is a selected position follows from the exit condition of the `while` loop. If one of them is a selected position, the minimality of its index follows from using $m = \min(\mathsf{next\_sel}[i], \mathsf{next\_sel}[j])$ to bound the value of ℓ throughout the algorithm. This concludes the proof of the correctness of Algorithm 1.

We now turn to proving that Algorithm 1 runs in time $O(t)$. We use the following properties to bound the number of loop iterations.

Lemma 6.3.4. *In Algorithm 1 of Algorithm 1, either $T[i+\ell] \nsim T[j+\ell]$, or (at least) one of $T[i+\ell], T[j+\ell]$ is a selected position or a wildcard.*

Proof. In Algorithm 1, r is the LCE of $T_\#[i+\ell..n]$ and $T_\#[j+\ell..n]$, and therefore $T_\#[i+\ell+r] \neq T_\#[j+\ell+r]$. Then, in Algorithm 1 the value of ℓ is set to either $\ell + r$ or m.

In the former case, we either have $T[i+\ell] \nsim T[j+\ell]$ (and we are done) or $T[i+\ell] \sim T[j+\ell]$, and one of $T[i+\ell], T[j+\ell]$ is a wildcard as $T_\#[i+\ell+r] \neq T_\#[j+\ell+r]$.

In the latter case, i.e., if ℓ is set to $m = \min(\mathsf{next\_sel}[i], \mathsf{next\_sel}[j])$ in Algorithm 1, then, by the definition of next_sel, at least one of $T[i+\ell], T[j+\ell]$ is a selected position. □

Lemma 6.3.5. *In Algorithm 1 of Algorithm 1, either $T[i+\ell] \nsim T[j+\ell]$, or (at least) one of $T[i+\ell], T[j+\ell]$ is a selected position or a transition position.*

Proof. By Lemma 6.3.4, we have that in Algorithm 1 of Algorithm 1, either $T[i+\ell] \nsim T[j+\ell]$, or at least one of $T[i+\ell], T[j+\ell]$ is a selected position or a wildcard. We consider three sub-cases.

If $T[i+\ell] \nsim T[j+\ell]$ in Algorithm 1, then neither $T[i+\ell]$ nor $T[j+\ell]$ is a wildcard, and d is 0 in Algorithm 1. Hence, the value of ℓ does not change.

If one of $T[i+\ell], T[j+\ell]$ is a selected position in Algorithm 1, then we have $\ell = m$ by the minimality of m, and ℓ will be set to the same value in Algorithm 1 regardless of the value of d.

Finally, assume that one of $T[i+\ell], T[j+\ell]$ is a wildcard in Algorithm 1. Then for any $p \in \{i+\ell, j+\ell\}$ such that $T[p]$ is a wildcard, $T[p+\mathsf{next\_tr}[p]]$ is a transition position (by the definition of $\mathsf{next\_tr}$). Before executing Algorithm 1, d is the maximum of these $\mathsf{next\_tr}[p]$, and hence one of $T[i+\ell+d]$ or $T[j+\ell+d]$ is a transition position. □

By Lemma 6.3.5, the number of transition positions between $i+\ell$ or $j+\ell$ and the corresponding next selected position decreases by at least one (or the algorithm exits the loop and returns). The use of m in Lines 6 and 12 ensures that we cannot go over a selected position, and, by construction, there are at most t transition positions between i or j and the next selected position, therefore Algorithm 1 goes through at most $2t$ iterations of the loop. Each iteration consists of one LCE query in $T_\#$ and a constant number of constant-time operations, hence Algorithm 1 takes time $O(t)$ overall given a data structure for constant-time LCE queries on $T_\#$.

LCEW query algorithm. Let ℓ denote the result of Algorithm 2 on some input (i, j). The properties of Algorithm 1 ensure that $T[i..i+\ell) \sim T[j..j+\ell)$. As the algorithm returns when it either encounters a mismatch or reaches the end of the string, the matching fragment cannot be extended, which ensures the maximality of ℓ. To prove that Algorithm 2 runs in time $O(t)$, we show that it makes a constant number of loop iterations.

Algorithm 2 Algorithm to answer the query $\mathsf{LCEW}(i, j)$

```
1:  function Query(i, j)
2:      ℓ ← 0
3:      while i + ℓ ≤ n and j + ℓ ≤ n do
4:          ℓ ← NextSelectedOrMismatch(i + ℓ, j + ℓ)
5:          if T[i + ℓ] ≁ T[j + ℓ] then
6:              return ℓ
7:          if i + ℓ ∈ Sel then
8:              ℓ ← ℓ + Jump[i + ℓ, j + ℓ] + 1
9:          else
10:             ℓ ← ℓ + Jump[j + ℓ, i + ℓ] + 1
11:     return ℓ
```

Lemma 6.3.6. *The `while` loop of Algorithm 2 makes at most three iterations.*

Proof. After the call to Algorithm 1 in Algorithm 2, either $T[i+\ell] \nsim T[j+\ell]$ or at least one of $i+\ell, j+\ell$ is a selected position. In the former case, this is the last iteration of the loop. In the latter case, suppose without loss of generality that $i+\ell$ is a selected position. Then, in Algorithm 2 the value of ℓ is updated to $\textsc{Jump}[i+\ell, j+\ell]+1$, and there is no selected position between $i+\ell$ and the end of the longest common extension

with wildcards. This can happen at most once for each of i and j, and thus the loop goes through at most three iterations before exiting. □

Intuitively, the first call to NextSelectedOrMismatch finds the last selected position in $T[i..i+\ell)$ or in $T[j..j+\ell)$, the second call finds the last selected position in the other fragment, and the last call finds either a mismatch or the end of the text. Therefore, Algorithm 2 makes up to three calls to Algorithm 1 plus a constant number of operations, and thus runs in time $O(t)$.

6.4 Connection to Boolean Matrix Multiplication

In this section, we describe a fine-grained connection between LCEW and (sparse) Boolean matrix multiplication. In Section 6.4.1, we use this connection to obtain a lower bound on the preprocessing-query-time product of combinatorial data structures for LCEW. In Section 6.4.2, we further connect sparse matrices and strings with few groups of wildcards, deriving an efficient multiplication algorithm.

6.4.1 A Lower Bound for Combinatorial Data Structures

Our lower bound is based on the combinatorial matrix multiplication conjecture which states that for any $\varepsilon > 0$ there is no combinatorial algorithm for multiplying two $n \times n$ Boolean matrices working in time $O(n^{3-\varepsilon})$. Gawrychowski and Uznanski [158, Conjecture 3.1] showed that the following formulation is equivalent to this conjecture:

Conjecture 6.4.1 (Combinatorial matrix multiplication conjecture)**.** *For every $\varepsilon > 0$ and $\alpha, \beta, \gamma > 0$, there is no combinatorial algorithm that computes the product of Boolean matrices of dimensions $n^\alpha \times n^\beta$ and $n^\beta \times n^\gamma$ in time $O(n^{\alpha+\beta+\gamma-\varepsilon})$.*

Theorem 6.4.2. *Under Conjecture 6.4.1, there is no combinatorial data structure that solves the LCEW problem with preprocessing time $O(n^a)$ and query time $O(n^b)$, where a and b are fixed real numbers (independent of n), $a \geq b \geq 0$, and $a + b < 2 - \varepsilon$ for some constant $\varepsilon > 0$.*

Proof. Assume for the sake of contradiction that there exists such a data structure. We can then use it to derive an algorithm that contradicts Conjecture 6.4.1.

Let α, β be positive constants, and let A and B be rectangular Boolean matrices of respective dimensions $p \times q$ and $q \times p$, where $p = n^\alpha, q = n^\beta$ and $\beta = c \cdot \alpha$ (i.e., $q = p^c$) for a constant c to be fixed later. We encode A into a string S_A of length $pq = p^{c+1}$ in row-major order, that is, $S_A[qi+j+1] = \phi_A(A[i+1, j+1])$ for $i \in [0..p), j \in [0..q)$, and B into a string S_B of length $qp = p^{c+1}$ in column-major order, that is, $S_B[i+qj+1] = \phi_B(B[i+1, j+1])$ for $i \in [0..q), j \in [0..p)$, where:

$$\phi_A(x) = \begin{cases} 1 & \text{if } x = 1 \\ \Diamond & \text{if } x = 0 \end{cases} \quad \text{and} \quad \phi_B(y) = \begin{cases} 2 & \text{if } y = 1 \\ \Diamond & \text{if } y = 0 \end{cases}$$

It follows from this definition that $\phi_A(x)$ does not match $\phi_B(y)$, i.e., $\phi_A(x) \nsim \phi_B(y)$, if and only if $x = y = 1$.

▷ Claim 6.4.3. We have $(AB)[i, j] = 1 \iff \mathsf{LCEW}_{S_A,S_B}(qi+1, qj+1) < q$.

Claim proof. Having $\mathsf{LCEW}_{S_A,S_B}(qi+1, qj+1) < q$ means that there exists an index $k < q$ such that $S_A[qi+1+k]$ does not match $S_A[qj+1+k]$. By construction, $S_A[qi+k+1]$ does not match $S_B[k+qj+1]$ if and only if $A[i,k] = B[k,j] = 1$. Now, $(AB)[i,j] = 1$ if and only if there exists an index $k \leq q$ such that $A[i,k] = B[k,j] = 1$, which concludes the proof. ◁

Therefore, we can compute each entry of $C = AB$ using one LCEW query between S_A and S_B. To perform LCEW queries between S_A and S_B, we instantiate our LCEW data structure on the string S_AS_B, which has length $2pq = O(p^{c+1})$, and therefore has $G = O(p^{c+1})$ consecutive wildcard groups. Computing C then takes $O(p^2 \cdot p^{b(c+1)})$ time for the queries, plus $O(p^{a(c+1)})$ time to build the data structure.

Assume first that $a > b$. We set $c = 2/(a-b)-1 > 0$ to balance the terms of the above two expressions, and obtain an algorithm that computes C in time $O(p^{2a/(a-b)})$. This contradicts Conjecture 6.4.1 if this running time is $O(p^{(2+c-\delta)})$ for some constant $\delta > 0$. In terms of exponents, this happens when $2a/(a-b) < 2+c-\delta$. Let $\delta = \varepsilon/(a-b) > 0$. We have:

$$\begin{aligned} a+b < 2-\varepsilon &\Leftrightarrow 2a < a-b+2-\varepsilon \\ &\Leftrightarrow \frac{2a}{a-b} < 1 + \frac{2}{a-b} - \frac{\varepsilon}{a-b} \\ &\Leftrightarrow \frac{2a}{a-b} < 2 + \frac{2}{a-b} - 1 - \delta \\ &\Leftrightarrow \frac{2a}{a-b} < 2 + c - \delta \end{aligned} \tag{6.2}$$

Eq. (6.2) results in a contradiction for all $a > b$ that satisfy $a + b < 2 - \varepsilon$. Assume now that $a = b$. We set $c = 2/\delta$ for $\delta = \varepsilon/2$ and obtain:

$$2 + b(1+c) < 2 + (1-\delta)(1+2/\delta) = (2+c)(1-\delta/2) \leq 2 + c - \delta/2 \tag{6.3}$$

We hence obtain the desired contradiction for $a = b$ with $a + b < 2 - \varepsilon$ as well, thus concluding the proof. □

6.4.2 Fast Sparse Matrix Multiplication

In this section, we further connect sparse matrices and strings with few groups of wildcards, deriving an algorithm for sparse Boolean matrix multiplication (BMM).

Observation 6.4.4. *Let A and B be sparse matrices with m_{in} ones in total. The string S_AS_B contains $G \leq m_{in} + 1$ groups of consecutive wildcards.*

Furthermore, our reduction to LCEW can also exploit the sparsity of the output matrix C. The algorithm underlying the proof of Theorem 6.4.2 uses $pr = p^2$ LCEW queries to compute C, one query for each entry. When the output matrix contains at most m_{out} non-zero values, we show that we can reduce the number of LCEW queries to $2p+m_{out}-1$, using the following lemma.

Lemma 6.4.5. *Let t be an integer, and let $i, j < p-t$. We have $\mathsf{LCEW}_{S_A,S_B}(qi+1, qj+1) \geq q \cdot t$ if and only if for every $x < t$, $(AB)[i+x, j+x] = 0$.*

Proof. First, if $\mathsf{LCEW}_{S_A,S_B}(qi+1, qj+1) \geq q \cdot t$, then for every $x < t$, $\mathsf{LCEW}_{S_A,S_B}(q(i+x)+1, q(j+x)+1) \geq q$. By Claim 6.4.3, this implies that $(AB)[i+x, j+x] = 0$. The converse follows from the fact that $\mathsf{LCEW}(i,j) \geq a$ and $\mathsf{LCEW}(i+a, j+a) \geq b$ implies that $\mathsf{LCEW}(i,j) \geq a+b$ for any integers $a, b \geq 0$. □

The above lemma readily implies that the answer to an LCEW query at the first indices of a diagonal gives us the length of a longest prefix run of zeroes in this diagonal. A repeated application of this argument implies that computing the entries in the d-th diagonal of C takes $m_d + 1$ LCEW queries, where m_d is the number of non-zero entries in this diagonal. Summing over all $2p - 1$ diagonals, this gives a total of $2p - 1 + m_{out}$ queries. As a corollary, we obtain the following algorithm for the multiplication of sparse matrices.

Theorem 6.4.6. *Let A and B be $n \times n$ sparse Boolean matrices such that A and B contain m_{in} non-zero entries and $C = (AB)$ contains m_{out} non-zero entries. There is a deterministic combinatorial algorithm that computes C in time $O(n\sqrt{m_{in} \cdot (n + m_{out})} \log^2 n)$.*

Proof. We assume that the matrices are given as a list of coordinates of 1 bits, sorted by row index and then by column index: this compact representation has size $O(m_{in})$.

We can assume without loss of generality that $m_{in} \geq n$, otherwise we can remove the empty rows and columns (i.e., rows and columns with zeroes only) from A and B (an empty row in A induces an empty row in C, while an empty column in B induces an empty column in C), and pad the output with zeroes where necessary. (For sparse matrices, this means offsetting the indices of the non-zeroes.) This procedure takes $O(m_{in} + m_{out})$ time overall.

Consider the string $S = S_A S_B$ defined in the proof of Theorem 6.4.2, which has length $2n^2$ and contains $G \leq m_{in} + 1$ groups of wildcards, and let $t = n\sqrt{G/(n + m_{out})}$. If $t \leq G$, we instantiate the data structure of Theorem 6.1.2 for S with this parameter t, and if $t > G$, we use the data structure of Corollary 6.1.3. This is always possible as $t \leq n\sqrt{G} \leq G \cdot |S|^{1/2}$. Then, using the argument described just above this theorem, we can compute C using $2n + m_{out} - 1$ LCEW queries.

For $t \leq G$, construction takes time $O(n^2(G/t) \cdot \log n) = O(n\sqrt{m_{in} \cdot (n + m_{out})} \log n)$ and answering the queries takes total time $O((n + m_{out}) \cdot t) = O(n\sqrt{m_{in} \cdot (n + m_{out})})$. In the other case, constructing the data structure takes $O(n^2)$ time, and answering the queries takes the same time as in the first case.

Accounting for the preprocessing to ensure that $m_{in} \geq n$, the total running time is $O(n^2 + n\sqrt{m_{in} \cdot (n + m_{out})} \log n + m_{in} + m_{out})$. As $n \leq m_{in} \leq n^2$ and $m_{out} \leq m_{in}^2$, the $n\sqrt{m_{in} \cdot (n + m_{out})}$ term is asymptotically larger than $n^2 + m_{in} + m_{out}$. This yields the desired runtime.

Choosing t requires knowing an estimate of m_{out}: if it is not known, we estimate it using binary search between 1 and n^2. For a given estimate of m_{out}, we run the algorithm, halting and restarting whenever the total query time exceeds the construction time. This search adds an extra $O(\log n)$ factor in the time complexity. □

6.5 Faster Approximate Pattern Matching and Computation of Periodicity Arrays construction

In this section, we use the data structure of Theorem 6.1.2 to derive improved algorithms for the k-PMWE problem and the problem of computing periodicity arrays of strings with wildcards.

6.5.1 Faster Pattern Matching with Errors and Wildcards

We first consider the problem of pattern matching with errors, where both the pattern and the text may contain wildcards. Recall that the edit distance between two strings X, Y, denoted by $\mathsf{ed}(X, Y)$, is the smallest number of insertions, deletions, and substitutions of a character, required to transform X into Y. The problem is formally defined as follows:

Problem 6.5.1 (k-PMWE)**.**
▷ **Input:** A text T of length n, a pattern P of length m and an integer threshold k.
▷ **Output:** Every position p for which there exists $i \leq p$ such that $\mathsf{ed}(T[i..p], P) \leq k$.

Akutsu [20] gave an algorithm for this problem that runs in time $\tilde{O}(n\sqrt{km})$. Building upon their framework, we show that the complexity can be reduced to $O(n(k + \sqrt{kG \log m}))$, where G is the cumulative number of groups of wildcards in P and T (or equivalently, the number of groups of wildcards in $P\$T$).[2]

Theorem 6.5.2. *There is an $O(n(k + \sqrt{kG \log m}))$-time algorithm for k-PMWE.*

Proof. Akutsu [20, Proposition 1] shows that, if after an α-time preprocessing, LCEW queries between P and T can be answered in time $\beta \geq 1$, then the k-PMWE problem can be solved in time $O(\alpha + n\beta k)$.

First, assume that $G \log m \geq k$. We use the data structure of Theorem 6.1.2 with $t = \sqrt{(G/k) \cdot \log m} \geq 1$ to answer LCEW queries: we then have $\alpha = O(n\sqrt{Gk \log m})$ (here we use the standard trick to replace the $\log n$ factor in the construction time with $\log m$, namely, if $n \geq 2m$, we divide T into n/m blocks of length $\leq 2m$ overlapping by m characters, and build such a data structure for each block independently) and $\beta = O(t) = O(\sqrt{(G/k) \cdot \log m})$. Therefore, the running time of the algorithm is $O(n\sqrt{Gk \log m})$. Second, if $G \log m < k$, we simply set $t = 1$: the total running time is then $O(nG \log m + nk) = O(nk)$. Accounting for both cases, the time complexity of this algorithm is $O(n(k + \sqrt{Gk \log m}))$. □

6.5.2 Faster Computation of Periodicity Arrays

Our data structure also enables us to obtain efficient algorithms for computing periodicity arrays of a string with wildcards (Theorem 6.5.3). These algorithms build on and improve upon the results of Iliopoulos and Radoszewski [180].

Theorem 6.5.3. *Let S be a string of length n with G groups of wildcards. The prefix array, the quantum and deterministic border arrays and the quantum and deterministic period arrays of S can be computed in $O(n\sqrt{G} \log n)$ time and $O(n)$ space.*

By Fact 6.2.2, it remains to show that the prefix array of S can be computed in $O(n\sqrt{G} \log n)$ time and $O(n)$ space. Recall that the *prefix array* of a string S of length n is an array π of size n such that $\pi[i] = \mathsf{LCEW}(1, i)$. Consequently, π can be computed using n LCEW queries in S. By instantiating our data structure with $t = \sqrt{G}$, we obtain an algorithm running in $O(n\sqrt{G} \log n)$ time, but its space usage is $\Theta(n\sqrt{G})$. Below, we show how one can slightly modify the data structure of Theorem 6.1.2 to reduce the space complexity to $O(n)$, extending the ideas of Iliopoulos and Radoszewski [180].

[2] The additive "k" term in our complexity is necessary because $G \log m$ might be smaller than k. On the other hand, one can assume w.l.o.g. that $m \geq k$. Hence this additional term is hidden in the runtime of Akutsu's algorithm [20].

Lemma 6.5.4. *Let S be a string of length n with G groups of wildcards. The prefix array of S can be computed in $O(n\sqrt{G}\log n)$ time and $O(n)$ space.*

Proof. We add the index 1 to the set of selected positions Sel and preprocess S in $O(n)$ time and space to support LCE queries on $S_{\#}$ in $O(1)$ time [138].

Notice that, using the dynamic programming algorithm of Lemma 6.3.2, for any $r < \lambda$, the row $(\textsc{Jump}[i_r, j], j = 1, \ldots, n)$ of the $\textsc{Jump}$ table can be computed in time $O(n \log n)$ and $O(n)$ space from the next row $(\textsc{Jump}[i_{r+1}, j], j = 1, \ldots, n)$. It suffices to compute the array A_r of occurrences of P_r in time $O(n \log n)$ using the algorithm of Clifford and Clifford [100], and then apply the recurrence relation of Eq. (6.1).

To answer an $\mathsf{LCEW}(1, j)$ query, we perform the following steps: first, we issue a $\textsc{Jump}[1, j]$ query followed by at most t regular LCE queries. If after this process we reach a mismatch or the end of S, we are done. Otherwise, we need to perform another $\textsc{Jump}$ query from indices (ℓ_j, i_{r_j}), where $i_{r_j} = j + \ell_j - 1$ is a selected position, and then perform at most t more regular LCE queries from the resulting positions. We store the indices (ℓ_j, i_{r_j}) for every value j, grouped by selected position i_{r_j}.

To answer the first batch of $\textsc{Jump}[1, j]$ queries, we use the above observation iterated $\lambda - 1$ times to compute the first row of the $\textsc{Jump}$ table in $O(n(G/t) \cdot \log n)$ time and $O(n)$ space. To answer the next batch of queries (i.e., $\textsc{Jump}[\ell_j, i_{r_j}]$ queries) in $O(n(G/t) \cdot \log n)$ time and $O(n)$ space, we again use the above observation to iterate over the rows of the $\textsc{Jump}$ table, starting from row i_λ and going up, storing only one row at a time. After computing the row corresponding to a selected position i_r, we answer all $\textsc{Jump}$ queries with $i_{r_j} = i_r$ and then perform the remaining LCE queries to answer the corresponding $\mathsf{LCEW}(1, j)$ query.

The claimed bounds follow by setting $t = \sqrt{G} \leq G$. □

Part II

Approximate Language Membership

Chapter 7

Overview

7.1 Property Testing: Superfast Approximate Decision Procedures

Introduced in 1998 by Goldreich, Goldwasser, and Ron [166], Property Testing studies the amount of information needed to "analyze" an input X, e.g. to compute a function F on X, or to decide whether it has a given property P. In this thesis, we focus on the latter task: we are given an input X of length n and have to decide which of $P(X)$ and $\neg P(X)$ holds. However, it turns out that, in almost all cases, distinguishing between the two cases requires having almost complete information about X, i.e. reading $\Omega(n)$ bits of X. Therefore, most efforts have focused on testing properties *approximately*: the task becomes deciding whether X has the property P or whether X is *far* from having P. Consequently, Property Testing is often described as the field of "super-fast approximate decision procedures" [165, Preface]. The notion of being "far" from a property depends on the task at hand and is usually parameterized by a real $\varepsilon > 0$: then, X is ε-far from P if one needs to change at least an ε-fraction of X to obtain an object Y that has property P.

Property testing has been studied for a wide variety of objects. The idea was first introduced as a tool for program checking, with applications to locally-testable error-correcting codes and probabilistically checkable proofs (PCPs) [73, 258]. Properties of statistical distributions are a natural candidate for randomized approximate testing, and also received a lot of interest [54, 55, 82, 282]. Other lines of work studied testing properties of Boolean functions, such as juntas [135] and linearity [247], and of graphs [22, 166], such as triangle-freeness [26], or k-colorability [21]. Alon et al. [24], followed by others [136, 142, 231, 248] considered testing membership in formal (regular and context-free) languages: in Chapter 8 of this thesis, we follow the same line of work.

7.1.1 Queries: measuring "information complexity"

The "amount of information" needed to decide the property is formally defined by *query complexity*. In the property testing framework, the input is initially hidden, and the algorithm can make *queries* to an oracle to reveal information about the input. The information returned by the oracle depends on the input object and the task. For statistical distributions, a query usually returns a sample from the distribution. For discrete objects such as graphs or strings, a query reveals a specific part of the input. For example, in graphs, the oracle may reveal whether two given vertices are adjacent. In string problems, such as the one studied in Chapter 8, a query on a given index i returns the letter at that

position in the input. The *query complexity* of an algorithm A for a property testing task is the function that, given n, measures the number of queries A makes to the input, in the worst case over all inputs of length n. The main line of research in Property Testing focuses on understanding the exact asymptotic query complexity of testing a property, both in terms of upper and lower bounds.

The motivation for query complexity arises from situations where reading the input is expensive, so the algorithm should access it as infrequently as possible. For example, this may be due to latency when the input is stored in distributed remote storage (such as AWS cloud services [122]), or due to computational cost when the input must be computed via an API call or decoded.

It turns out that, in many cases, by a clever combination of random sampling and algorithmic decisions, one can test a property with a number of queries that is *sublinear* in n, and sometimes even with a *constant* number of queries. A landmark result in graph property testing is the result of Alon and Shapira [23, Theorem 1], who showed that *every* monotone graph property is testable with a *constant* number of queries, i.e. a number of queries that does not depend on the size of the input graph (but it does depend on the proximity parameter ε). Alon, Fischer, Newman, and Shapira [25] later gave an *exact* characterization of the graph properties that can be tested with a constant number of queries. Property Testing results usually come with and are based on combinatorial results that reveal useful structure in ε-far instances (see e.g. [21, 24]). These results show that such instances contain many "obstructions" to the property, and that random sampling can find one with high probability. For example, in the case of triangle-freeness of graphs, the result of Alon et al. [26] is based on the Triangle Removal Lemma of Ruzsa and Szemerédi [260], which shows that any graph on n vertices that is ε-far from being triangle-free contains at least $\delta_\varepsilon n^3$ triangles, where δ_ε is a constant that depends only on ε, but not on n.

7.1.2 Testing formal languages

In 2001, Alon, Krivelevich, Newman, and Szegedy [24] initiated the study of property testing of formal languages. In this task, one is given a language L, and the goal is to decide whether the input x is in L, or if it is ε-far from L. Being "ε-far from L" is defined with respect to a metric d between strings, which depends on the context, usually the Hamming distance or the edit distance (in what follows, we assume the Hamming distance, unless explicitly stated otherwise). Here, "x is ε-far from L" means that the distance $d(x, L)$ between x and L is at least εn, where n is the length of L. The distance between x and the language L is defined as the minimum distance between x and an element of L, or $+\infty$ if L is empty:

$$d(x, L) = \min_{y \in L} d(x, y).$$

Alon et al. [24] study algorithms for testing membership in regular and context-free languages, using the Hamming distance for d. For regular languages, they show an upper bound of $O(\log^3(\varepsilon^{-1})/\varepsilon)$ queries and a lower bound of $\Omega(1/\varepsilon)$ queries for a large class of *non-trivial languages*. On the other hand, they exhibit a context-free language for which membership testing requires $\Omega(\sqrt{n})$ queries. Building on their work, Magniez and de Rougemont [231] gave a tester using $O(\log^2(\varepsilon^{-1})/\varepsilon)$ queries for regular languages under the "edit distance with moves", and François et al. [142] gave a tester using $O(1/\varepsilon^2)$ queries for the weighted edit distance. For context-free languages, subsequent work has

considered testing specific context-free languages such as the DYCK languages [136, 248] or regular tree languages [231].

7.1.3 The Complexity of Testing Regular Languages

The results of Alon, Krivelevich, Newman, and Szegedy [24] on the complexity of testing membership in a regular language leave a gap between the upper bound of $O(\log^3(\varepsilon^{-1})/\varepsilon)$ queries and the lower bound of $\Omega(1/\varepsilon)$ queries. In [48], T. Starikovskaya and I showed that any regular language can be tested using $O(\log(\varepsilon^{-1})/\varepsilon)$ queries, and that *there exists* a regular language L_0 that requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries, closing the complexity gap.

Alon et al. [24] remarked that some regular languages, which they call *trivial languages*, are much easier to test: for sufficiently large n, the answer depends only on the *input length* n, and not on the input itself, therefore they can be tested without querying the input. Alon et al. [24] also showed that testing a non-trivial language requires $\Omega(1/\varepsilon)$ queries. However, one can observe that many non-trivial languages can be tested with $O(1/\varepsilon)$ queries, in contrast to the harder language L_0 which requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries. In summary, some regular languages are easier to test than others, and the above observations lead to the definition of three classes of "trivial", "easy", and "hard" languages, with respective query complexity 0, $\Theta(1/\varepsilon)$ and $\Theta(\log(\varepsilon^{-1})/\varepsilon)$. This raises the following two questions:

1. Are there other complexity classes of regular languages, e.g. languages with complexity $\Theta(\frac{1}{\varepsilon}\log\log(1/\varepsilon))$?
2. Can we give an effective characterization of the languages in each class?

In Chapter 8, we answer both questions. Building on the work of Alon et al. [24] and François et al. [142], we define the set MBS of *minimal blocking sequences* of a regular language, and show that this set gives an exact characterization of the complexity of testing a regular language L:

- L is *trivial* if and only if $\mathsf{MBS}(L)$ is empty,
- L is *easy* if and only if $\mathsf{MBS}(L)$ is finite but non-empty, and
- L is *hard* if and only if $\mathsf{MBS}(L)$ is infinite.

In particular, this shows that there are no other complexity classes. To obtain the result for trivial languages, we show that languages with an empty MBS are exactly the trivial languages identified by Alon et al. [24]. To separate easy and hard languages, we establish a structural connection between L and $\mathsf{MBS}(L)$: if L is recognized by an NFA with m states, then $\mathsf{MBS}(L)$ is also a regular language and is recognized by an NFA of size $2^{O(m)}$. Therefore, if $\mathsf{MBS}(L)$ is infinite, we can use the Pumping Lemma [254] to extract from $\mathsf{MBS}(L)$ an infinite language L' of minimal blocking sequences with a very repetitive structure. Using this language L', we can replicate the argument that gives a lower bound for L_0, showing that L is also hard. Incidentally, the characterization of $\mathsf{MBS}(L)$ as a regular language shows that our characterization is effective, in the sense that membership in each of the three classes is decidable in polynomial space; we further show that these problems are complete for PSPACE.

The work presented in this chapter previously appeared in two articles: one with T. Starikovskaya [48], which was published at ICALP'21, and one with C. Mascle and N. Fijalkow, accepted for publication at ICALP'25.

7.2 Distance to a Language, in Small Space

In Chapter 9, we study the problem of computing *how far* an input X is from having the target property, a natural extension of deciding whether X has the property P. When we consider the language L_P that represents P, this problem is known as the *language distance problem*. Introduced in the early 1970s by Aho and Peterson [18], the language distance problem has been studied extensively for regular languages under Hamming and edit distances [57], for general context-free languages, focusing mainly on the edit distance [18, 79, 98, 228, 242, 261], and the Dyck language (the language of well-nested parentheses sequences) [11, 43, 79, 98, 120].

7.2.1 Small space algorithms

While the algorithms in the above works often have an asymptotically optimal time complexity, their space usage is often prohibitively large. For example, the CYK algorithm[1] and its derivatives use space quadratic in the input size. Therefore, recent research efforts have focused on designing algorithm with a lower space usage, often in the streaming or in the read-only model. Babu et al. [41] gave streaming algorithms for deciding membership in the DLIN and LL(k) subclasses of context-free languages. Ganardi et al. [153] studied the space complexity of recognizing context-free languages in the *sliding window* model, a variant of the streaming model where one must decide whether the string formed by the last n characters of the text is in a given language. Ganardi [149] later considered the same problem for the subclass of Visibly Pushdown languages, and Ganardi et al. [155] gave low-latency algorithms that extend these results.

7.2.2 Low distance regime

In some cases, we only want to know whether an input is *close* to the language, and if so, how close. This is formalized using the "low-distance regime": we are given a threshold parameter k, and either report that the distance is greater than k, or report the distance if it does not exceed k. In this framework, many problems have solutions that are more efficient than for the general case, with a complexity usually parameterized by k. For example, the best known algorithm for computing the edit distance between two strings of length n takes time $O(n^2/\log n)$ [237], and under the Strong Exponential Time Hypothesis (SETH), this cannot be improved [42] to $O(n^{2-\varepsilon})$ for any $\varepsilon > 0$. On the other hand, in the low-distance regime, we can compute the edit distance with threshold k in time $O(n + k^2)$ [226]. Similarly, the best algorithm for computing the Hamming distance between a pattern P of length m and all substrings of length m of a text T of length n takes $O(n\sqrt{m})$ time [86], which is improved to $O(n\sqrt{k \log k})$ in the low-distance regime [32].

7.2.3 Computing the distance to palindromes and squares, online and in small space

In Chapter 9, we study the complexity of the *online* and *low-distance* version of the language distance problem, with an emphasis on small-space algorithms. We consider both the edit distance and the Hamming distance, and focus on two classical languages: the language PAL of all palindromes, where a palindrome is a string that is equal to its

[1] See Section 1.2.

reversed copy, and the language SQ of all squares, where a square is the concatenation of two copies of a string. These two languages are very similar and yet very different in nature: PAL is not regular but context-free, while SQ is not even context-free. Amir and Porat [31] gave a randomized streaming algorithm that solves the problem of computing the k-bounded distance to PAL in $\tilde{O}(k)$ space and $\tilde{O}(k^2)$ time per input character. Continuing their line of research, we give streaming algorithms using $\mathrm{poly}(k, \log n)$ time per character and $\mathrm{poly}(k, \log n)$ space for all four problems. As a corollary, building on the work of Gawrychowski et al. [161], we obtain new streaming algorithms for approximating the maximal length of a substring of a given text that is close to PAL in a text.

While streaming algorithms are extremely efficient (in particular, the above space complexities account for *all* the space used by the algorithms, including the space needed to store information about the input), they are inherently randomized, which means that there is a small probability that they will produce incorrect results. Motivated by this, we also study the problems in the read-only model: for the four problems, we show *deterministic* algorithms that use $\mathrm{poly}(k, \log n)$ time per character and $\mathrm{poly}(k, \log n)$ *extra* space (not accounting for the input). As a side result of independent interest, we develop the first *deterministic* read-only algorithms for computing k-mismatch and k-edit occurrences of a pattern in a text using $\mathrm{poly}(k, \log n)$ space.

All the results in this chapter use similar techniques. First, we show that the distance (Hamming or edit) from a string U to PAL or SQ can be expressed in terms of the distance between substrings of U or its reversal. Using small-space distance sketches (of Clifford et al. [107] for the Hamming distance, and of Bhattacharya and Koucký [61] for the edit distance), we obtain streaming algorithms for computing the distance to PAL. By combining these sketches with streaming approximate pattern matching algorithms and an *ad hoc* filtering procedure, we obtain algorithms for computing the distance to SQ. By replacing the streaming pattern matching algorithms with read-only algorithms, either off-the-shelf or custom, we obtain small-space read-only algorithms for both problems.

The results presented in this chapter appeared in an article published at ISAAC'23 [49], co-authored with T. Kociumaka and T. Starikovskaya.

7.3 Palindromic Length: a Different Notion of Proximity

In the previous section, we discussed algorithms for computing the (Hamming or edit) distance between a string X and a language L, as a measure of how far X is from the property defined by L. However, this approach does not take into account the *semantic* meaning of L. For example, consider PAL, the language of palindromes: a *palindrome* is a non-empty string that reads the same forward and backward, e.g. $X = abba$ is a palindrome. The string $Y = a^n b^n = aa \dots abb \dots b$ is at distance n from the set of palindromes, but it can be written as the concatenation of two palindromes a^n and b^n. In this sense, Y is *close* to being a palindrome, and this notion of proximity is hard to capture with a metric over strings. The *palindromic length* extends this notion to all strings: the palindromic length of the string Y is the smallest integer p such that Y can be written as the concatenation of p palindromes.

7.3.1 Previous work on algorithms for palindromic length and related problems

The problem of computing palindromic length was first studied in the low-distance regime. In terms of formal languages, this means deciding whether the input is in the language PAL^k. Galil and Seiferas [148] gave linear-time recognition algorithms for the cases $k = 1, 2, 3, 4$, but the general question remained open for almost 40 years. Only in 2015, Kosolobov et al. [216] showed an $O(nk)$-time recognition algorithm for PAL^k for all positive k, which was finally improved to the optimal $O(n)$ time by Rubinchik and Shur [257] in 2020 .

For computing the palindromic length of a string T, multiple independent $O(n \log n)$-time algorithms were presented in [133, 179, 256]. More recently, Borozdin, Kosolobov, Rubinchik, and Shur [76] showed an optimal $O(n)$-time algorithm for this problem.

Since one-letter strings are all palindromes, any string can be written as the concatenation of *some number* of palindromes and has a finite palindromic length. This property does not hold for the languages even-length palindromes $\mathrm{PAL}_{\mathrm{ev}}$ and the language of palindromes of length greater than one $\mathrm{PAL}_{>1}$, which are natural derivations of PAL. The problem of recognizing the languages $\mathrm{PAL}^*_{\mathrm{ev}}$ (often referred to as "palstar"), $\mathrm{PAL}^*_{>1}$, is a classical question of formal language theory, initiated in the seminal paper of Knuth, Morris, and Pratt [201]. In the early 1970s, it was widely believed that $\mathrm{PAL}^*_{\mathrm{ev}}$ could not be recognised in linear time, and it was considered as a candidate for a "hard" context-free language [201, Section 6]. However, Knuth, Morris, and Pratt [201] refuted this hypothesis by showing an $O(n)$-time recognition algorithm for $\mathrm{PAL}^*_{\mathrm{ev}}$. Manacher [234] found another way to recognize $\mathrm{PAL}^*_{\mathrm{ev}}$ in linear time, and Galil [145] derived a real-time recognition algorithm (see also Slisenko [270]). Later, Galil and Seiferas [148] showed a linear-time recognition algorithm for $\mathrm{PAL}^*_{>1}$.

7.3.2 Small-space algorithms for palindromic length in the low-distance regime

All known near-linear-time algorithms for computing the palindromic length of a string use linear space. In Chapter 10, we focus on the *space complexity* of recognizing PAL^k and computing the palindromic length of a string, and give a small-space algorithm for computing the palindromic length in the low-distance regime.

Our algorithm is based on a more versatile data structure that space-efficiently encodes the prefixes of a text that are in PAL^k, called k-palindromic prefixes; this data structure uses $6^{O(k^2)} \cdot \log^k n$ space. The linear-time algorithm of Borozdin et al. [76] for computing the palindromic length is based on a combinatorial characterization of the structure of palindromic prefixes of a text: they have a highly repetitive structure and can be partitioned into $O(\log n)$ arithmetic progressions. Our data structure is based on a novel higher-order extension of this idea: we show that the set of k-palindromic prefixes of a string can be partitioned into a small number of *affine prefix sets*, a high-order extension of arithmetic progressions that can be stored compactly. We also show that our characterization leads to efficient algorithms for constructing the encoding of k-palindromic prefixes. Our algorithm for computing the palindromic length of a string is a by-product of this data structure.

We also give an information-theoretic lower bound on the space required to encode the k-palindromic prefixes of a text of length n, showing that our data structure cannot be drastically improved.

The results presented in this chapter resulted in an article co-authored with J. Ellert and T. Starikovskaya, available as a preprint on the ArXiv [53].

Chapter 8

The Complexity of Testing Regular Languages

8.1 Introduction

Property testing was introduced by Goldreich, Goldwasser, and Ron [166] in 1998: it is the study of randomized approximate decision procedures that must distinguishing objects that have a given property of those that are *far* from having it. Because of this relaxation on the input object, the field focuses on very efficient decision procedures, typically with sublinear (or even constant) running time – in particular, the algorithm does not even have the time to read the whole input.

In a seminal paper, Alon, Krivelevich, Newman, and Szegedy [24] introduced property testing of formal languages: given a language L (a set of finite words), the goal is to determine whether an input word u belongs to the language or is ε-far[1] from it, where ε is the precision parameter. The model assumes random access to the input word: a *query* specifies a position in the word and asks for the letter at that position, and the *query complexity* of the algorithm is the worst-case number of queries it makes to the input. Alon et al. [24] showed a surprising result: under the Hamming distance, all regular languages are testable with $O(\log^3(\varepsilon^{-1})/\varepsilon)$ queries, where the $O(\cdot)$ notation hides constants that depend on the language, but, crucially, not on the length of the input word. In that paper, they also identified the class of *trivial* regular languages, those for which the answer only depends on the length n of the input (and not on the content of the input word) for sufficiently large n, e.g. finite languages or the set of words starting with an a. They showed that testing membership in a *non-trivial* regular language requires $\Omega(1/\varepsilon)$ queries.

The results of Alon et al. [24] leave a gap of $O(\log^3(1/\varepsilon))$ between the best upper and lower bounds. We set out to improve our understanding of property testing of regular languages by closing this gap. With Tatiana Starikovskaya, we obtained in 2021 [48] the first improvement over the result of Alon et al. [24] in more than 20 years:

Theorem 8.1.1 (From [48, Theorem 5])**.** *Under the edit distance, every regular language can be tested with $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

Testers under the edit distance are weaker than testers under the Hamming distance, hence this result does not exactly improve the result of Alon et al. [24]. We overcome this

[1] Informally, u is ε-far from L means that even by changing an ε-fraction of the letters of u, we cannot obtain a word in L. See Section 8.2 for a formal definition.

shortcoming later in this chapter: Theorem 8.4.16 extends the above result to the case of the Hamming distance.

We also showed that this upper bound is tight, in the sense that there is a regular language L_0 for which this complexity cannot be further improved, thereby closing the query complexity gap.

Theorem 8.1.2 (From [48, Theorem 15]). *There is a regular language L_0 with query complexity $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ under the edit distance[2], for all small enough $\varepsilon > 0$.*

Furthermore, it is easy to find specific non-trivial regular languages for which there is an algorithm using only $O(1/\varepsilon)$ queries, e.g. $L = a^*$ over the alphabet $\{a, b\}$, $L = (ab)^*$ or $L = (aa + bb)^*$.

Hence, these results combined with those of Alon et al. [24] show that there exist trivial languages (that require 0 queries for large enough n), *easy* languages (with query complexity $\Theta(1/\varepsilon)$) and *hard* languages (with query complexity $\Theta(\log(\varepsilon^{-1})/\varepsilon)$). This raises the question of whether there exist languages with a different query complexity (e.g. $\Theta(\log\log(\varepsilon^{-1})/\varepsilon)$), or if every regular language is either trivial, easy or hard. This further asks the question of giving a characterization of the languages that belong to each class, inspired by the recent success of exact characterizations of the complexity of sliding window recognition [152] and dynamic membership [29] of regular languages.

In this chapter, we answer both questions: we show a trichotomy of the complexity of testing regular languages under the Hamming distance, showing that there are only the three aforementioned complexity classes (trivial, easy and hard), we give a characterization of all three classes using a combinatorial object called *blocking sequences*, and show that this characterization can be decided in polynomial space (and that it is complete for PSPACE).

8.1.1 Overview of the chapter

In this section, we give a high-level overview of the approach and notions used for proving the results of this chapter. This sections assumes familiarity with classical notions such as finite automata; the formal definitions of all concepts used in this section can be found in Section 8.2.

Let us start with the notion of a property tester for a language L: the goal is to determine whether an input word u belongs to the language L, or whether it is ε-far from it. We say that u of length n is *ε-far from L* with respect to a metric d over words if all words $v \in L$ satisfy $d(u, v) \geq \varepsilon n$, written $d(u, L) \geq \varepsilon n$. Throughout this work and unless explicitly stated otherwise, we will consider the case where d is the Hamming distance, defined for two words u and v as the number of positions at which they differ if they have the same length, and as $+\infty$ otherwise. In that case, $d(u, L) \geq \varepsilon n$ means that one cannot change a proportion ε of the letters in u to obtain a word in L. We assume random access to the input word: a query specifies a position in the word and asks for the letter in this position. A *ε-property tester* (or for short, simply a *tester*) T for a language L is a randomized algorithm that, given an input word u of length n, always answers "yes" if $u \in L$ and answers "no" with probability bounded away from 0 when u is ε-far from L. The measure of complexity of a tester that focus we on in this chapter is its *query complexity*, which is the maximum number of queries that T makes to an input of length

[2]Note that, as opposed to testers, lower bounds for the edit distance are stronger than lower bounds of the Hamming distance.

n, as a function of n and ε, in the worst case over all words of length n and all possible random choices.

We can now formally define the classes of *trivial, easy* and *hard* regular languages.

Definition 8.1.3 (Hard, easy and trivial languages)**.** *Let L be a regular language. We say that:*

- *L is* hard *if the optimal query complexity for a property tester for L is $\Theta(\log(\varepsilon^{-1})/\varepsilon)$.*
- *L is* easy *if the optimal query complexity for a property tester for L is $\Theta(1/\varepsilon)$.*
- *L is* trivial *if there exists $\varepsilon_0 > 0$ such that for all positive $\varepsilon < \varepsilon_0$, there is a property tester and some $n_0 \in \mathbb{N}$ such that the tester makes 0 queries on words of length $\geq n_0$.*

Remark 8.1.4. If L is finite, then it is trivial: since there is a bound B on the lengths of its words, a tester can reject words of length at least $n_0 = B + 1$ without querying them. For that reason, we only consider *infinite* languages in the rest of the chapter.

Our characterization of those three classes uses the notion of *blocking sequence* of a language L. Intuitively, they are sequences of words such that finding those words as factors of a word w proves that w is not in L. We also define a partial order on them, which gives us a notion of *minimal* blocking sequence.

Theorem 8.1.5. *Let L be an infinite regular language recognized by an NFA $\mathcal{A}$ and let $\mathsf{MBS}(\mathcal{A})$ denote the set of minimal blocking sequences of $\mathcal{A}$. The complexity of testing L is characterized by $\mathsf{MBS}(\mathcal{A})$ as follows:*

1. *L is trivial if and only if $\mathsf{MBS}(\mathcal{A})$ is empty;*
2. *L is easy if and only if $\mathsf{MBS}(\mathcal{A})$ is finite and nonempty;*
3. *L is hard if and only if $\mathsf{MBS}(\mathcal{A})$ is infinite.*

In the case where L is recognized by a strongly connected automaton, blocking sequences can be replaced by *blocking factors*. A blocking factor is a single word that is not a factor of any word in L.

Section 8.2 defines the necessary terms and notations. The rest of the chapter is structured as follows. In Sections 8.3 and 8.4, we delimit the set of hard languages, that is, the ones that require $\Theta(\log(\varepsilon^{-1})/\varepsilon)$ queries.

More precisely, Section 8.3 focuses on the subcase of languages defined by *strongly connected automata*. First, we combine the ideas of Alon et al. [24] with those that we presented in [48] to obtain a property tester that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries for any language with a strongly connected automaton, under the Hamming distance. Second, we show that if the language of a strongly connected automaton has infinitely many blocking factors then it requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries. This result generalizes the result that we previously gave in [48], which was for a single language, to all regular languages with infinitely many minimal blocking factors. We use Yao's minimax principle [286]: this involves constructing a hard distribution over inputs, and showing that any deterministic property testing algorithms cannot distinguish between positive and negative instances against this distribution.

In Section 8.4, we extend those results to all automata. The interplay with the previous section is different for the upper and the lower bound. For the upper bound of $O(\log(\varepsilon^{-1})/\varepsilon)$ queries, we use a natural but technical extension of the proof in the strongly connected case. Note that this result is an improvement over the result that we obtained in [48], which works under the edit distance, and testers for the Hamming distance are also testers for the edit distance. For the lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries for languages

with infinitely many minimal blocking sequences, we reduce to the strongly connected case. The main difficulty is that it is not enough to consider strongly connected components in isolation: there exists finite automata that contain a strongly connected component that induces a hard language, yet the language of the whole automaton is easy. We solve this difficulty by carefully defining the notion of minimality for a blocking sequence.

Section 8.5 completes the trichotomy, by characterizing the easy and trivial languages. We show that languages of automata with finitely many blocking sequences can be tested with $O(1/\varepsilon)$ queries. We also prove that if an automaton has at least one blocking sequence, then it requires $\Omega(1/\varepsilon)$ queries to be tested, by showing that the languages that our notion of trivial language coincides with the *trivial* languages of Alon et al. [24]. By contrast, we show that automata without blocking sequences recognize trivial languages.

Once we have the trichotomy, it is natural to ask whether it is effective: given an automaton $\mathcal{A}$, can we determine if its language is trivial, easy or hard? The answer is yes, and we show in Section 8.6 that all three decision problems are PSPACE-complete, even for strongly connected automata.

8.1.2 Related work

Building upon the seminal work of Alon et al. [24], Magniez and de Rougemont [231] gave a tester using $O(\log^2(\varepsilon^{-1})/\varepsilon)$ queries for regular languages under the edit distance with moves, and François et al. [142] gave a tester using $O(1/\varepsilon^2)$ queries for the case of the weighted edit distance. Alon et al. [24] also show that context-free languages cannot be tested with a constant number of queries, and subsequent work has considered testing specific context-free languages such as the DYCK languages [136, 248] or regular tree languages [231]. Property testing of formal languages has been investigated in other settings: Ganardi et al. [154] studied the question of testing regular languages in the so-called "sliding window model", while François et al. [142] considered property testing for Visibly Pushdown languages in the streaming model.

8.2 Preliminaries

Words and languages.

In this chapter, words are usually denoted with lowercase letters $u, v, \ldots$, and their letters are indexed starting at 0. To avoid collision with the precision threshold ε, the empty word is denoted γ. We use $w \preccurlyeq u$ to denote "w is a factor of u". Furthermore, if w is a factor of u and $w \neq u$, we say that w is a *proper factor* of u.

Finite automata.

Definition 8.2.1 (Nondeterministic Finite automaton)**.** *A* nondeterministic finite automaton *(NFA) $\mathcal{A}$ is a transition system defined by a tuple $(Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ the transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states.*

The *size* of an automaton, denoted $|\mathcal{A}|$, is its number of states. We say that $\mathcal{A}$ is deterministic (resp. complete [249, Section 2.4]) if $|\delta(q, a)| \leq 1$ (resp. ≥ 1) for all $q \in Q$ and $a \in \Sigma$. We say that there is a *transition*, or a *run*, from $p \in Q$ to $q \in Q$ labeled by $w \in \Sigma^*$, denoted $p \xrightarrow{w} q$, if there exist states $p_0, p_1, \ldots, p_{|w|}$ such that $p_0 = p, p_{|w|} = q$,

and for every $i = 0, \ldots, |w| - 1$, $p_{i+1} \in \delta(p_i, w[i])$. In this case, we say that q is reachable from p and that p is co-reachable from q. The *language recognized by* $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words that label a transition from the initial state to a final state, i.e.

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q_f \in F : q_0 \xrightarrow{w} q_f\}.$$

We say that an NFA is *trim* if every state is reachable from the initial state and co-reachable from some final state [249, Section 2.4]. An NFA $\mathcal{A}$ can always be converted into a trim NFA that recognizes the same language by removing the states of $\mathcal{A}$ that are either not reachable from q_0 or not co-reachable from any final state. Therefore, in this chapter, we assume w.l.o.g. that the NFA $\mathcal{A}$, which describes the language that we test, is trim.

To illustrate some of our examples and results, we use the classical representation of finite automata as state machines, as depicted in Fig. 8.1.

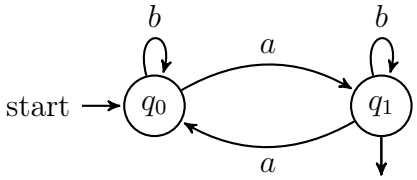

Figure 8.1: Graphical representation of a finite automaton. Circles represent the states of the automaton: the initial state is marked by an arrow coming from the word "start" and accepting states are marked by an outgoing arrow. The transition function is represented using arrows between states: if $q \in \delta(p, a)$, then there is an arrow from p to q with the label a. This finite automaton recognize the language of words with an odd number of a's.

Property testing.

Definition 8.2.2. *Let L be a language, let u be a word of length n, let $\varepsilon > 0$ be a precision parameter and let $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{+\infty\}$ be a metric. We say that the word u is* ε-far from L w.r.t. d *if $d(u, L) \geq \varepsilon n$, where*

$$d(u, L) := \min_{v \in L} d(u, v).$$

Throughout this work and unless explicitly stated otherwise, we will consider the case where d is the Hamming distance, defined for two words u and v as the number of positions at which they differ if they have the same length, and as $+\infty$ otherwise. In that case, $d(u, L) \geq \varepsilon n$ means that one cannot change an ε-fraction of the letters in u to obtain a word in L.

Definition 8.2.3. *A property tester for the language L with precision $\varepsilon > 0$ is a randomized algorithm T that, for any input u of length n, given random access to u, satisfies the following properties:*

$$\text{if } u \in L, \text{ then } T(u) = 1,$$
$$\text{if } u \text{ is } \varepsilon\text{-far from } L, \text{ then } \mathbb{P}\left(T(u) = 0\right) \geq 2/3.$$

The query complexity of T is a function of n and ε that counts the maximum number of queries that T makes over all inputs of length n and over all possible random choices.

Note that the above definition corresponds to property testers with *perfect completeness*: they always accept positive instances. Because they are based on the notion of blocking factors that we will discuss below, all known testers for regular languages [24, 48, 142, 231] have perfect completeness.

In this chapter, we assume that the automaton $\mathcal{A}$ that describes the tested language L is *fixed*, and not part of the input. Therefore, we consider its number of states m as a constant.

Graphs and periodicity.

We now recall tools introduced by Alon et al. [24] to deal with periodicity in finite automata.

Let $G = (V, E)$ with $E \subseteq V^2$ be a directed graph. A *strongly connected component* (or SCC) of G is a maximal set of vertices that are all reachable from each other. It is *trivial* if it contains a single state with no self-loop on it. We say that G is *strongly connected* if its entire set of vertices is an SCC.

The period $\lambda = \lambda(G)$ of a graph G is the greatest common divisor of the length of the cycles in G. If G is acyclic, we set $\lambda(G) = \infty$. Following the work of Alon et al. [24], we will use the following property of directed graphs.

Fact 8.2.4 (From [24, Lemma 2.3])**.** *Let $G = (V, E)$ be a nonempty, strongly connected directed graph with finite period $\lambda = \lambda(G)$. Then there exists a partition $V = Q_0 \sqcup \ldots \sqcup Q_{\lambda-1}$ and a reachability constant $\rho = \rho(G)$ that does not exceed $3|V|^2$ such that:*

1. *For every $0 \leq i, j \leq \lambda - 1$ and for every $u \in Q_i, v \in Q_j$, the length of any directed path from u to v in G is equal to $(j - i) \mod \lambda$.*
2. *For every $0 \leq i, j \leq \lambda - 1$, for every $u \in Q_i, v \in Q_j$ and for every integer $r \geq \rho$, if $r = (j - i) \pmod \lambda$, then there exists a directed path from u to v in G of length r.*

The sets $(Q_i : i = 0, \ldots, \lambda - 1)$ are the *periodicity classes* of G. In what follows, we will slightly abuse notation and use Q_i for arbitrary non-negative integers i to mean $Q_{i \pmod \lambda}$ when $i \geq \lambda$.

Given a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, we can naturally obtain the underlying directed graph by removing the labels from the transitions: it is the graph $G = (Q, E)$ where $E = \{(p, q) \in Q^2 \mid \exists a \in \Sigma : q \in \delta(p, a)\}$. In what follows, we naturally extend the notions of period[3], reachability constant and periodicity classes to finite automata through this graph G. Note that the numbering of the periodicity classes is defined up to a shift mod λ: we can thus always assume that Q_0 is the class that contains the initial state q_0. Similarly, we say that a finite automaton is strongly connected if the underlying graph is strongly connected. A *strongly connected component* (SCC for short) S of an automaton $\mathcal{A}$ is a maximal subset of states such that every state of S is reachable from every other one. Its *period* $\lambda(\mathcal{A})$ is the period of the graph G.

Positional words and positional languages.

Consider the language $L_1 = (ab)^*$, recognized by the automaton depicted in Fig. 8.2. The word $v = ab$ can appear as a factor of a word $u \in L_1$ if v occurs at an even position in u. However, if v occurs at an *odd* position in u, then u cannot be in L_1. Therefore, v can be

[3]Note that in this context, an *aperiodic automaton* means an automaton with an aperiodic underlying graph, which is not the same thing as a counter-free automata, which are sometimes called aperiodic automata.

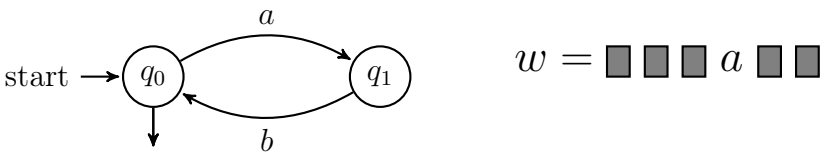

Figure 8.2: An automaton $\mathcal{A}$ that recognizes the language $L_1 = (ab)^*$. A witness that a word is not in this language is an a on an odd position or a b on an even position, hence w is not in $\mathcal{L}(\mathcal{A})$.

used to witness that u is not in L_1, but only if we find it at an odd position. This example leads us to introducing *p-positional words*, which additionally encode information about the index of each letter modulo an integer p.

More generally, we will associate to each regular language a period λ, and working with λ-positional words will allow us to define blocking factors in a position-dependent way without explicitly considering the index at which the factor occurs.

Definition 8.2.5 (Positional words)**.** *Let p be a positive integer. A* p-positional word *is a word over the alphabet $\mathbb{Z}/p\mathbb{Z} \times \Sigma$ of the form $(n \pmod p, a_0)((n+1) \pmod p, a_1) \cdots ((n+\ell) \pmod p, a_\ell)$ for some non-negative integer n. If $u = a_0 \cdots a_\ell$, we write $(n : u)$ to denote this word.*

With this definition, if $u = abcd$ and we consider the 2-positional word $\tau = (0 : u)$, the factor bc appears at position 1 in u and is mapped to the factor $\mu = (1, a)(0, b)$. In this case, even when taking factors of μ, we still retain the (congruence classes of the) indices in the original word τ.

Any strongly connected finite automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ can naturally be extended into an automaton over $\lambda(\mathcal{A})$-positional words as follows. Let $Q_0, \ldots, Q_{\lambda-1}$ be the partition of the states of $\mathcal{A}$ given by Fact 8.2.4, where $\lambda = \lambda(\mathcal{A})$ is the period of $\mathcal{A}$. The *positional extension of* $\mathcal{A}$ is the finite automaton $\widehat{\mathcal{A}}$ defined by:

$$\widehat{\mathcal{A}} = (Q, \mathbb{Z}/\lambda\mathbb{Z} \times \Sigma, \delta', q_0, F), \text{ where } \delta'(q, (i, a)) = \begin{cases} \delta(q, a) & \text{if } q \in Q_i, \\ \emptyset & \text{otherwise.} \end{cases}$$

By Fact 8.2.4, any transition from a state in Q_i goes to a state in Q_{i+1}, hence $\widehat{\mathcal{A}}$ recognizes well-formed λ-positional words.

We call the language recognized by $\widehat{\mathcal{A}}$ the *positional language of* $\mathcal{A}$, and denote it $\mathcal{PL}(\mathcal{A})$. This definition is motivated by the following property:

Property 8.2.6. *For any word $u \in \Sigma^*$, we have $u \in \mathcal{L}(\mathcal{A})$ if and only if $(0 : u) \in \mathcal{PL}(\mathcal{A})$.*

Positional words make it easier to manipulate factors with positional information, hence we phrase our property testing results in terms of positional languages. Notice that a property tester for $\mathcal{PL}(\mathcal{A})$ immediately gives a property tester for $\mathcal{L}(\mathcal{A})$, as one can simulate queries to $(0 : u)$ with queries to u by simply pairing the index of the query modulo $\lambda(\mathcal{A})$ with its result.

8.3 Hard Languages for Strongly Connected NFAs

Before considering the case of arbitrary NFAs, we first study the case of strongly connected NFAs, which are NFAs such that for any pair of states $p, q \in Q$, there exists a word w

such that $p \xrightarrow{w} q$. We will later generalize the results of this section to all NFAs.

We show that the query complexity of the language of such an NFA $\mathcal{A}$ can be characterized by the cardinality of the set of *minimal blocking factors* of $\mathcal{A}$, which are factor-minimal $\lambda(\mathcal{A})$-positional words that witness the fact that a word does not belong to $\mathcal{PL}(\mathcal{A})$. In this section, we consider a fixed NFA $\mathcal{A}$ and simply use "positional words" to refer to λ-positional words, where $\lambda = \lambda(\mathcal{A})$ is the period of $\mathcal{A}$.

Definition 8.3.1 (Blocking factors)**.** *Let $\mathcal{A}$ be a strongly connected NFA. A positional word τ is a* blocking factor *of $\mathcal{A}$ if for any positional word μ, we have $\tau \preccurlyeq \mu \Rightarrow \mu \notin \mathcal{PL}(\mathcal{A})$.*

Further, we say that τ is a minimal *blocking factor of $\mathcal{A}$ if no proper factor of τ is a blocking factor of $\mathcal{A}$. We use $\mathsf{MBF}(\mathcal{A})$ to denote the set of all minimal blocking factors of $\mathcal{A}$.*

Intuitively and in terms of automata, the positional word $(i : u)$ is blocking for $\mathcal{A}$ if there is no transition in $\mathcal{A}$ labeled by u that starts from a state of Q_i. (This property is formally established later in Lemma 8.3.5.) The main result of this section is the following:

Theorem 8.3.2. *Let L be an infinite language recognized by a strongly connected NFA $\mathcal{A}$. If $\mathsf{MBF}(\mathcal{A})$ is infinite, then L is hard, i.e., it has query complexity $\Theta(\log(\varepsilon^{-1})/\varepsilon)$.*

This result is both an upper bound of $O(\log(\varepsilon^{-1})/\varepsilon)$ queries and a lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries on the complexity of a tester for L: we prove the upper bound in Section 8.3.2 and the lower bound in Section 8.3.3.

8.3.1 Positional words, blocking factors and strongly connected NFAs

We first establish some properties of positional words that will help us ensure that we are creating well-formed positional words, that is, positional words where the index i of a letter $(i : a)$ is equal to $j + 1 \pmod{\lambda}$, where j is the index of the previous letter. In Section 8.3.2, we highlight the connection between property testing and blocking factors in strongly connected NFAs.

We start with the following properties, which are consequences of Fact 8.2.4.

Corollary 8.3.3. *Let n be a nonnegative integer, let w be a word of length n. If for some states $p \in Q_i, q \in Q_j$ of $\mathcal{A}$ we have $p \xrightarrow{w} q$, then the indices i, j satisfy the equation*

$$j - i = |w| \pmod{\lambda}$$

Corollary 8.3.4. *Let $\tau = (i : u)$ and $\mu = (j : v)$ be positional words. If $\tau \preccurlyeq \mu$, then there exists positional words η, η' with $|\eta| = i - j \pmod{\lambda}$ such that $\mu = \eta\tau\eta'$. In particular, this implies that there exists words w, w' with $|w| = i - j \pmod{\lambda}$ such that $v = wuw'$.*

These properties allows us to formalize the intuition we gave earlier about blocking factors.

Lemma 8.3.5. *A positional word $\tau = (i : u)$ is a blocking factor for $\mathcal{A}$ iff for every states $p \in Q_i, q \in Q$, we have $p \not\xrightarrow{u} q$.*

Proof. We first show that if there exists states $p \in Q_i, q \in Q$ such that $p \xrightarrow{u} q$, then τ is not blocking, i.e. there exists $\mu \in \mathcal{PL}(\mathcal{A})$ such that $\tau \preccurlyeq \mu$. As $\mathcal{A}$ is strongly connected, there exist positional words η, η' such that $q_0 \xrightarrow{\eta} p$ and $q \xrightarrow{\eta'} q_f$ for some $q_f \in F$. By Fact 8.2.4,

the positional word $\mu = \eta\tau\eta'$ is well formed. Furthermore, it labels a transition from q_0 to q_f, hence it is in $\mathcal{PL}(\mathcal{A})$, and τ is not blocking.

For the converse, assume that τ is non-blocking: we show that there exists two states $p \in Q_i, q \in Q$ such that $p \xrightarrow{u} q$. As τ is non-blocking, there exists a positional word $\mu = (0 : w)$ such that $\tau \preccurlyeq \mu$ and there exists a final state r such that $q_0 \xrightarrow{\mu} r$, and equivalently, $q_0 \xrightarrow{w} r$. By Corollary 8.3.4, since $\tau \preccurlyeq \mu$, there exists words v, v' such that $w = vuv'$ and the length of v is equal to i modulo λ. In particular, the path $q_0 \xrightarrow{w} r$ can be decomposed into $q_0 \xrightarrow{v} p \xrightarrow{u} q \xrightarrow{w} r$, and we have $p \xrightarrow{u} q$. It only remains to show that p is in Q_i: this follows by Corollary 8.3.3 since $|v| = i \pmod{\lambda}$. □

Next, we show that the Hamming distance between u and $\mathcal{L}(\mathcal{A})$ is the same as the (Hamming) distance between $(0 : u)$ and $\mathcal{PL}(\mathcal{A})$.

▷ Claim 8.3.6. For any word $u \in \Sigma^*$, we have $d(u, \mathcal{L}(\mathcal{A})) = d((0 : u), \mathcal{PL}(\mathcal{A}))$.

Claim proof. The $\leq$ part is straightforward. For the reverse inequality, if suffices to see that in any minimal substitution sequence from $(0 : u)$ to a positional word in $\mathcal{PL}(\mathcal{A})$, no operation changes only the index of an (index, letter) pair. ◁

The above claim allows us to interchangeably use the statements "u is ε-far from $\mathcal{L}(\mathcal{A})$" and "$(0 : u)$ is ε-far from $\mathcal{PL}(\mathcal{A})$".

8.3.2 An efficient property tester for strongly connected NFAs.

In this section, we show that for any strongly connected NFA $\mathcal{A}$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.

Theorem 8.3.7. *Let $\mathcal{A}$ be a strongly connected NFA. For any $\varepsilon > 0$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

Our proof is similar to the one we gave in [48], with one notable technical improvement: we use a new method for sampling factors in u, which greatly simplifies the correctness analysis.

8.3.2.1 An efficient sampling algorithm

We first introduce a sampling algorithm (Algorithm 3) that uses few queries and has a large probability of finding at least one factor from a large set $\mathcal{S}$ of disjoint "special" factors. Using this algorithm on a large set of disjoint blocking factors gives us an efficient property tester for strongly connected NFAs. We will re-use this sampling procedure later in the case of general NFAs (Theorem 8.4.16).

The procedure is fairly simple: the algorithm samples factors of various lengths in u at random. On the other hand, the correctness of the tester is far from trivial. The lengths and the number of factors of each length are chosen so that the number of queries is minimized and the probability of finding a "special" factor is maximized, regardless of their repartition in u. (In what follows, the "special" factors are blocking factors.)

▷ Claim 8.3.8. A call to SAMPLER(u, N, L) (Algorithm 3) makes $O(n \log(L)/N)$ queries to u.

Algorithm 3 Efficient generic sampling algorithm

```
1: function OneSample(u, ℓ)
2:     i ← uniform(0, n − 1)
3:     l ← max(i − ℓ, 0), r ← min(i + ℓ, n − 1)
4:     return u[l..r]
5: function Sampler(u, N, L)
6:     n ← |u|
7:     β ← n/N
8:     T ← ⌈log(L)⌉
9:     F ← ∅
10:    for t = 0 to T do
11:        ℓ_t ← 2^t, r_t ← ⌈2 ln(3)β/ℓ_t⌉
12:        for i = 0 to r_t do
13:            F ← F ∪ {OneSample(u, ℓ_t)}
14:    return F
```

Claim proof. A call to $\textsc{OneSample}(u, \ell_t)$ makes at most $2\ell_t$ queries to u. The function $\textsc{Sampler}(u, N, L)$ makes $r_t = 2\ln(3) \cdot \beta/\ell_t = 2\ln(3) \cdot n/(N\ell_t)$ calls to $\textsc{OneSample}(u, \ell_t)$ for each $t = 0, \ldots, T$, where $T = \lceil \log(L) \rceil$. This adds up to

$$\sum_{t=0}^{T} r_t \cdot \ell_t = \lceil \log(L) \rceil \cdot 2\ln(3) \cdot n/N = O(n \log(L)/N)$$

queries to u. ◁

Lemma 8.3.9. *Let u be a word of length n, and consider a set $\mathcal{S}$ containing at least N disjoint factors of u, each of length at most L. A call to the function* $\textsc{Sampler}(u, N, L)$ *(Algorithm 3) returns a set $\mathcal{F}$ of factors of u such that there exists a word of $\mathcal{S}$ that is a factor of some word of $\mathcal{F}$, with probability at least $2/3$.*

Proof. We conceptually divide the blocking factors in $\mathcal{S}$ into different categories depending on their length: let $T = \lceil \log(L) \rceil$, and for $t = 0, \ldots, T$, let S_t denote the subset of $\mathcal{S}$ which contains factors of length at most $\ell_t = 2^t$. We then carefully analyze the probability that randomly sampled factors of length $2\ell_t$ contains a factor from S_t, and show that over all t, at least one sampled factor contains a factor of $\mathcal{S}$, with probability at least $2/3$.

▷ Claim 8.3.10. If in a call to $\textsc{OneSample}$, the value i is such that there exists indices l and r such that $l \leq i \leq r$ and $u[l, r]$ contains a factor in $\mathcal{S}$, then the set $\mathcal{F}$ returned by the algorithm has the desired property.

As the factors given in $\mathcal{S}$ are disjoint, the probability p_t that the factor returned by $\textsc{OneSample}$ contains a factor from $\mathcal{S}$ is lower bounded by $p_t \geq \frac{1}{n} \sum_{v \in S_t} |v|$. The $\textsc{OneSample}$ function is called $r_t = 2\ln(3)\beta/\ell_t$ times independently for each t, hence the probability p that the algorithm samples a factor containing a factor from $\mathcal{S}$ satisfies the

following:

$$\begin{aligned}(1-p) = \prod_{t=0}^{T}(1-p_t)^{r_t} &\leq \exp\left(-\sum_{t=0}^{T} p_t r_t\right) \\ &\leq \exp\left(-\frac{2\ln(3)\beta}{n}\sum_{t=0}^{T}\frac{1}{\ell_t}\sum_{v\in S_t}|v|\right) \\ &= \exp\left(-\frac{2\ln(3)\beta}{n}\sum_{v\in\mathcal{S}}|v|\sum_{t=\lceil\log|v|\rceil}^{T}2^{-t}\right).\end{aligned}$$

Now, inverting the order of summation, and lower bounding the sum of powers of 2 by its first term, we obtain:

$$\begin{aligned}(1-p) &\leq \exp\left(-\frac{2\ln(3)\beta}{n}\sum_{v\in\mathcal{S}}|v|\cdot 2^{-\lceil\log|v|\rceil}\right) \\ &\leq \exp\left(-\frac{2\ln(3)\beta}{n}\sum_{v\in\mathcal{S}}|v|\frac{1}{2|v|}\right) \\ &= \exp\left(-\frac{2\ln(3)\beta}{n}\cdot\frac{|\mathcal{S}|}{2}\right) \leq \exp\left(-\frac{\ln(3)\beta N}{n}\right) \\ &= \exp\left(-\ln(3)\right) = 1/3\end{aligned}$$

It follows that $p \geq 2/3$, which concludes the proof. □

8.3.2.2 The tester

The algorithm for Theorem 8.3.7 is given in Algorithm 4.

Algorithm 4 Generic ε-property tester that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries

```
function TESTER(u, ε)
    n ← |u|, m ← |Q|
    L ← 12m²/ε
    if L(A) ∩ Σ^n = ∅ then
        Reject
    else if n < L then
        Query all of u and run A on it
        Accept if and only if A accepts
    else
        F ← SAMPLER((0 : u), n/L, L)
        Reject if and only if F contains a blocking factor for A.
```

We now show that Algorithm 4 is a property tester for $\mathcal{L}(\mathcal{A})$ that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries. In what follows, we use n to denote the length of the input word u and m to denote the number of states of $\mathcal{A}$.

▷ Claim 8.3.11. The tester given in Algorithm 4 makes $O(\log(\varepsilon^{-1})/\varepsilon)$ queries to u.

Proof. If $n \leq L$, then the tester makes $n \leq L = O(1/\varepsilon)$ queries, and the claim holds. Otherwise, the number of queries is given by the call to SAMPLER$(u, n/L, L)$: by Claim 8.3.8, this uses $O(\frac{n\log L}{n/L}) = O(L\log L) = O(\log(\varepsilon^{-1})/\varepsilon)$ queries. □

Alon et al. [24, Lemma 2.6] first noticed that if a word u is ε-far from $\mathcal{L}(\mathcal{A})$, then it contains $\Omega(\varepsilon n)$ short factors that witness the fact that u is not in $\mathcal{L}(\mathcal{A})$. We start by translating the lemma of Alon et al. on "short witnesses" to the framework of blocking factors. More precisely, we show that if u is ε-far from $\mathcal{L}(\mathcal{A})$, then $(0 : u)$ contains many disjoint (i.e. non-overlapping) blocking factors.

Lemma 8.3.12. *Let $\varepsilon > 0$, let u be a word of length $n \geq 6m^2/\varepsilon$ and assume that $\mathcal{L}(\mathcal{A})$ contains at least one word of length n. If $\tau = (0 : u)$ is ε-far from $\mathcal{PL}(\mathcal{A})$, then τ contains at least $\varepsilon n/(6m^2)$ disjoint blocking factors.*

Proof. We build a set $\mathcal{P}$ of disjoint blocking factors of τ as follows: we process u from left to right, starting at index $i_1 = \rho$, where ρ is the reachability constant of $\mathcal{A}$ (see Fact 8.2.4). Next, at iteration t, set j_t to be the smallest integer greater than or equal to i_t and smaller than $n - \rho$ such that $\tau[i_t..j_t]$ is a blocking factor. If there is no such integer, we stop the process. Otherwise, we add $\tau[i_t..j_t + \rho - 1]$ to the set $\mathcal{P}$, and iterate starting from the index $i_{t+1} = j_t + \rho$.

Let k denote the size of $\mathcal{P}$. We will show that we can substitute at most $3(k+1)m^2$ positions in τ to obtain a word in $\mathcal{PL}(\mathcal{A})$. (See Fig. 8.3 for an illustration of this construction.) Using the assumption that τ is ε-far from $\mathcal{PL}(\mathcal{A})$ (which follows from Claim 8.3.6) will give us the desired bound on k.

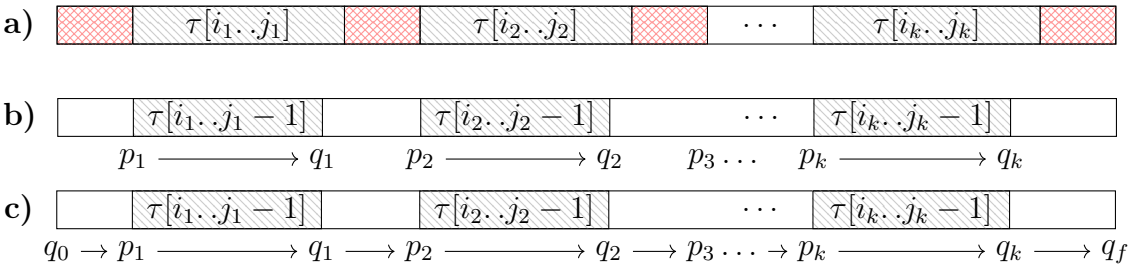

Figure 8.3: **a)** The decomposition process returns k factors $\tau[i_1, j_t], \ldots, \tau[i_k, j_k]$ (represented as diagonally hatched in gray regions), separated together and with the start of the text by padding regions of $\rho - 1$ letters (red crosshatched regions). **b)** If we exclude the last letter of each blocking factor, we obtain factors that label transitions between some pair of states p_t, q_t for each $t = 1, \ldots, k$. **c)** We use the padding regions to bridge between consecutive factors as well as the start and end of the word.

For every t, we chose j_t to be minimal so that $\tau[i_t..j_t]$ is blocking, hence $\tau[i_t..j_t - 1]$ is not blocking, and therefore $\tau[i_t..j_t - 1]$ labels a run from some state $p_t \in Q_{i_t}$ to some state $q_t \in Q_{j_t}$. Therefore, using the strong connectivity of $\mathcal{A}$ and Fact 8.2.4, we can substitute the letters in $\tau[j_t..j_t + \rho - 1]$ to obtain a factor that labels a transition from q_t to p_{t+1}. After this transformation, the word $\tau[i_t..j_t + \rho - 1]$ labels a transition from p_t to p_{t+1}. Using the ρ letters at the start and the end of the word, we add transitions from an initial state to p_1 and from q_k to a final state: the assumption that $\mathcal{L}(\mathcal{A})$ contains a word of length n ensures that Q_n contains a final state, hence this is always possible. The resulting word is in $\mathcal{PL}(\mathcal{A})$ and was obtained from τ using $(k+1)\rho \leq 3(k+1)m^2$ substitutions. As τ is ε-far from $\mathcal{PL}(\mathcal{A})$, we obtain the following bound on k:

$$\begin{aligned} 3(k+1)m^2 \geq \varepsilon n &\implies k \geq \frac{\varepsilon n}{3m^2} - 1 \\ &\implies k \geq \frac{\varepsilon n}{6m^2} \end{aligned}$$

The last implication uses the assumption that $n \geq 6m^2/\varepsilon$. □

Next, we show that if u is ε-far from $\mathcal{L}(\mathcal{A})$, then $(0 : u)$ contains $\Omega(\varepsilon n)$ blocking factors, each of length $O(1/\varepsilon)$.

Lemma 8.3.13. *Let $\varepsilon > 0$, let u be a word of length $n \geq 6m^2/\varepsilon$ and assume that $\mathcal{L}(\mathcal{A})$ contains at least one word of length n. If u is ε-far from $\mathcal{L}(\mathcal{A})$, then the positional word $(0 : u)$ contains at least $\varepsilon n/(12m^2)$ disjoint blocking factors of length at most $12m^2/\varepsilon$.*

Proof. Let $u, \mathcal{A}$ be a word and an automaton satisfying the above hypotheses. By Lemma 8.3.12, $(0 : u)$ contains at least $\varepsilon n/(6m^2)$ *disjoint* blocking factors. As these factors are disjoint, at most half of them (that is, $\varepsilon n/(12m^2)$ of them) can have length greater than $12m^2/\varepsilon$, as the sum of their lengths cannot exceed n. □

Proof of Theorem 8.3.7. First, note that if $u \in \mathcal{L}(\mathcal{A})$, $(0 : u)$ cannot contain a blocking factor for $\mathcal{A}$, hence Algorithm 4 always accepts u. Next, if $\mathcal{L}(\mathcal{A}) \cap \Sigma^n$ is empty or if $|u| \leq L = 12m^2/\varepsilon$, the tester has the same output as $\mathcal{A}$, hence it is correct.

In the remaining case, u is long enough and ε-far from $\mathcal{L}(\mathcal{A})$, hence Lemma 8.3.13 gives us a large set of short blocking factors in $(0 : u)$: this is exactly what the SAMPLER function needs to find at least one factor containing a blocking factor with probability at least $2/3$. More precisely, by Lemma 8.3.13, $(0 : u)$ contains at least $\varepsilon n/(12m^2) = n/L$ blocking factors of length at most $L = 12m^2/\varepsilon$, hence the conditions of Lemma 8.3.9 are satisfied.

As a factor containing a blocking factor is also a blocking factor, the set $\mathcal{F}$ computed on line 10 of Algorithm 4 contains at least one blocking factor with probability at least $2/3$, and Algorithm 4 satisfies Definition 8.2.3. □

8.3.3 Lower bound from infinitely many minimal blocking factors

We now show that languages with infinitely many *minimal* blocking factors are hard, i.e. any tester for such a language requires $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries.

Let us first give an example that will motivate our construction. Consider the parity language P consisting of words that contain an even number of b's, over the alphabet $\{a, b\}$. Distinguishing $u \in P$ from $u \notin P$ requires $\Omega(n)$ queries, as changing the letter at single position can change membership in P. However, P is trivial to test, as any word is at distance at most 1 from P, for the same reason. Now, consider language L_2 consisting of words over $\{a, b, c, d\}$ such that between a c and the next d, there is a word in P. Intuitively, this language encodes multiple instances of P, hence we can construct words ε-far from L_2, and each instance is hard to recognize for property testers, hence the whole language is. In [48, Theorem 15], we proved a lower bound of $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ on the query complexity of any property tester for L_2, matching the upper bound in the same paper.

The minimal blocking factors of L_2 include all words for the form cvd where $v \notin P$: there are infinitely many such words. This is no coincidence: we show in this thesis than this lower bound can be lifted to any language with infinitely many minimal blocking factors, under the Hamming distance.

Theorem 8.3.14. *Let $\mathcal{A}$ be a strongly connected NFA. If $\mathsf{MBF}(\mathcal{A})$ is infinite, then there exists a constant ε_0 such that for any $\varepsilon < \varepsilon_0$, any ε-property tester for $L = \mathcal{L}(\mathcal{A})$ uses $\Omega(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

The proof of this result is full generalization of the lower bound against the "repeated Parity" example given above.

Our proof is based on (a consequence of) Yao's Minimax Principle [286]: if there is a distribution $\mathcal{D}$ over inputs such that any *deterministic* algorithm that makes at most q queries errs on $u \sim \mathcal{D}$ with probability at least p, then any *randomized* algorithm with q queries errs with probability at least p on some input u.

To prove Theorem 8.3.14, we first exhibit such a distribution $\mathcal{D}$ for $q = \Theta(\log(\varepsilon^{-1})/\varepsilon)$. We take the following steps:

1. we show that with high probability, an input u sampled w.r.t. $\mathcal{D}$ is either in or ε-far from L (Lemma 8.3.21),
2. we show that with high probability, any deterministic tester that makes fewer than $c \cdot \log(\varepsilon^{-1})/\varepsilon$ queries (for a suitable constant c) cannot distinguish whether the instance u is positive or ε-far, hence it errs with large probability.
3. combine the above two results to prove Theorem 8.3.14 via Yao's Minimax principle.

8.3.3.1 The structure of $\mathsf{MBF}(\mathcal{A})$

Before diving into the proof of Theorem 8.3.14, we show that if $\mathsf{MBF}(\mathcal{A})$ is infinite, then we can find minimal blocking factors with a "regular" structure, a crucial ingredient for our proof. First, we prove that the set of minimal blocking factors of an automaton is a regular language, recognized by an automaton that is possibly exponentially larger than $\mathcal{A}$. We first prove the result for blocking factors of the form $(i : u)$ for a fixed $i \in \mathbb{Z}/\lambda\mathbb{Z}$.

Lemma 8.3.15. *Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a strongly connected NFA with m states and let $\lambda = \lambda(\mathcal{A})$. For every $i \in \mathbb{Z}/\lambda\mathbb{Z}$, the set of minimal blocking factors of $\mathcal{A}$ of the form $(i : u)$ is a regular language recognized by a NFA of size $2^{O(m)}$.*

Proof. We call blocking factors of $\mathcal{A}$ of the form $(i : u)$ its *i-blocking factors*.

We first show that the set of i-blocking factors of $\mathcal{A}$, but not necessarily minimal ones, is a regular language recognized by an NFA $\mathcal{A}_i$ with $m + 1$ states. The result follows by using standard constructions for complement and intersection of automata [249, Chapter 1, Section 3]: these constructions give an automaton of size $2^{O(m)}$ that recognizes words in L that have no proper factor in L.

Consider the NFA $\mathcal{A}_i$ obtained by adding a new sink state $\perp$ to $\mathcal{A}$, making it the only accepting state, with set of initial states Q_i. Formally, $\mathcal{A}_i$ is defined as $\mathcal{A}_i = (Q \cup \{\perp\}, \Sigma, \delta', Q_i, \{\perp\})$, where δ' is defined as follows:

$$\forall p \in Q, \forall a \in \Sigma : \delta'(p, a) = \begin{cases} \{\perp\} & \text{if } \delta(p, a) = \emptyset, \\ \delta(p, a) & \text{otherwise.} \end{cases}$$

This automaton[4] recognizes the set of i-blocking factors of $\mathcal{A}$ and has size $O(m)$. Applying the aforementioned construction to $L = \mathcal{L}(\mathcal{A}_i)$ yields the desired automaton, of size $2^{O(m)}$. □

It follows that the set of minimal blocking factors of $\mathcal{A}$ is also a regular language.

Corollary 8.3.16. *Let $\mathcal{A}$ be an NFA with m states. The set of minimal blocking factors of $\mathcal{A}$ is a regular language recognized by an NFA of size $2^{O(m)}$.*

[4] Our definition of NFAs does not allow for multiple initial states. As there is no constraint of strong connectivity for $\mathcal{A}_i$, this can be solved using a simple construction that adds a new initial state.

Therefore, if $\mathsf{MBF}(\mathcal{A})$ is infinite, we can use the Pumping Lemma [249, Chapter 1, Proposition 2.2] to find an infinite family of minimal blocking factors with a shared structure $\{\phi\nu^r\chi, r \in \mathbb{N}\}$, for some non-empty positional words ϕ, ν and χ. We will use this property later, when proving a lower bound against the language of automata with infinitely many blocking factors.

Lemma 8.3.17. *If $\mathsf{MBF}(\mathcal{A})$ is infinite, then there exist positional words ϕ, ν_+, ν_-, χ such that:*

1. *the words ν_+ and ν_- have the same length,*
2. *there exists a constant $S = 2^{\mathrm{poly}(m)}$ such that $|\phi|, |\nu_+|, |\nu_-|, |\chi| \leq S$,*
3. *there exists an index $i_* \in \mathbb{Z}/\lambda\mathbb{Z}$ and a state $q_* \in Q_{i_*}$ such that for every integer $r \geq 1$, the positional word $\tau_{-,r} = \phi(\nu_-)^r\chi$ is blocking for $\mathcal{A}$, and for every $s < r$, we have*
$$q_* \xrightarrow{\tau_{+,r,s}} q_* \text{ where } \tau_{+,r,s} = \phi(\nu_-)^j\nu_+(\nu_-)^{r-1-s}\chi.$$
In particular, $\tau_{+,r,s}$ is not blocking for $\mathcal{A}$.

Note that here, the state q_* is the same for *every* integers r, s.

Proof. As $\mathsf{MBF}(\mathcal{A})$ is infinite, there must exist an integer i_* such that $\mathcal{A}$ has infinitely many minimal i_*-blocking factors; we fix such i_* in what follows.

Let us recall the Pumping Lemma [249, Chapter 1, Proposition 2.2].

Fact 8.3.18 (Pumping Lemma)**.** *Let L be a regular language recognized by an automaton of size T. There exists an integer $S = O(T)$ such that any word u of L of length at least S can be factorized as $u = xyz$ such that $|xy| \leq S, |y| \geq 1$ and, for all $k \geq 0$, $xy^kz \in L$.*

As the set of minimal i_*-blocking factors is an infinite regular language recognized by an NFA of size $T = 2^{O(m)}$. Let $S = 2^{O(m)}$ be the constant given by the Pumping Lemma: since the language is infinite, it contains at least one positional word of length greater than S. Hence, there exist positional words τ, μ and η, with $|\mu| \geq 1$, such that for any non-negative integer $k, \tau\mu^k\eta$ is a minimal i_*-blocking factor. By removing factors that label loops in the automaton, we can assume that each of them has length at most S. Furthermore, we can assume w.l.o.g. that neither τ nor η is empty, otherwise we set their value to μ: after this modification, $\tau\mu^k\eta$ is still a minimal i_*-blocking factor for every $k \geq 0$.

Notice that the word $\tau\mu^m$ is not a blocking factor, as a proper factor of the minimal blocking factor $\tau\mu^m\eta$. Therefore, by the pigeonhole principle, there exist integers $k_0, k_1 \geq 1$ with $k_0 + k_1 = m$ and states p, p_1 such that we have

$$p \xrightarrow{\tau\mu^{k_0}} p_1 \xrightarrow{\mu^{k_1}} p_1.$$

Note that, by Fact 8.2.4, $p_1 \xrightarrow{\mu^{k_1}} p_1$ implies that $k_1 \cdot |\mu| = 0 \pmod{\lambda}$.

Similarly, the word $\mu^m\eta$ is not a blocking factor, since it is a proper factor of the minimal i_*-blocking factor $\tau\mu^m\eta$. Again, there exist integers $k_2 \geq 1, k_3$ summing to m and states p_2 and q such that

$$p_2 \xrightarrow{\mu^{k_2}} p_2 \xrightarrow{\mu^{k_3}\eta} q.$$

Now, define $\phi = \tau\mu^{k_0}, \chi = \mu^{k_3}\eta$ and $\nu_- = \mu^K$, where $K = \rho \cdot k_1 \cdot k_2$. As there are transitions starting from p_1 and p_2 labeled by μ, p_1 and p_2 belong to the same periodicity class. Therefore, by Fact 8.2.4, as $K \geq \rho$ and $K \cdot |\mu| = 0 \pmod{\lambda}$, there exists a word ν_+ of length $K \cdot |\mu|$ such that $p_1 \xrightarrow{\nu_+} p_2$. This choice of ϕ, ν_+, ν_- and χ satisfies all the conditions of the lemma. □

8.3.3.2 Constructing a Hard Distribution $\mathcal{D}$

Let $\varepsilon > 0$ be sufficiently small and let n be a large enough integer. In what follows, m denotes the number of states of $\mathcal{A}$. To construct the hard distribution $\mathcal{D}$, we will use an infinite family of blocking factors that share a common structure, given by Lemma 8.3.17.

The crucial property here is that $\tau_{-,r}$ and $\tau_{+,r,s}$ are very similar: they have the same length, differ in at most S letters, yet one of them is blocking and the other is not.

We now use the words $\tau_{-,r}$ and $\tau_{+,r,s}$ and the constant S to describe how to sample an input $\mu = (0 : u)$ of length n w.r.t. $\mathcal{D}$.

Let π be a uniformly random bit. If $\pi = 1$, we will construct a positive instance $\mu \in \mathcal{PL}(\mathcal{A})$, and otherwise the instance will be ε-far from $\mathcal{PL}(\mathcal{A})$ with high probability. We divide the interval $[0..n-1]$ into $k = \varepsilon n$ intervals of length $\ell = 1/\varepsilon$, plus small initial and final segments μ_i and μ_f of length $O(\rho)$ to be specified later. For the sake of simplicity, we assume that k and ℓ are integers and that λ divides ℓ. For $j = 1, \ldots, k$, let a_j, b_j denote the endpoints of the j-th interval. For each interval, we sample independently at random a variable κ_j with the following distribution:

$$\kappa_j = \begin{cases} t, & \text{with prob. } p_t = 3 \cdot 2^t S\varepsilon / \log((S\varepsilon)^{-1}) \text{ for } t = 1, 2, \ldots, \log((S\varepsilon)^{-1}), \\ 0, & \text{with prob. } p_0 = 1 - \sum_{t=1}^{\log((S\varepsilon)^{-1})} p_t. \end{cases} \tag{8.1}$$

The event $\kappa_j > 0$ means that the j-th interval is filled with $N \approx 2^{-\kappa_j}/\varepsilon$ "special" factors. When $\pi = 0$, these "special" factors will be minimal blocking factors $\tau_{-,r}$ for $r = 2^{\kappa_j}$, whereas when $\pi = 1$, they will instead be similar non-blocking factors $\tau_{+,r,s}$ for a uniformly random s: they will be hard to distinguish with few queries. On the other hand, the event $\kappa_j = 0$ means that the j-th interval contains no specific information. More precisely, we choose a positional word η_* of length ℓ such that $q_* \xrightarrow{\eta_*} q_*$: by Fact 8.2.4, this is possible as $\ell = 0 \pmod{\lambda}$. Then, if $\kappa_j = 0$, we set $\mu[a_j..b_j] = \eta_*$, regardless of the value of π.

Formally, if $\kappa_j > 0$, let $r = 2^{\kappa_j}$, $N = 2^{-\kappa_j}/(S\varepsilon)$ and let η be a word of length $\ell - N \cdot |\tau_{-,r}|$ such that $q_* \xrightarrow{\eta} q_*$: such a word exists as λ divides ℓ and $|\tau_{-,r}|$. We construct the j-th interval as follows:

- if $\pi = 0$, we set $\mu[a_j..b_j] = (\tau_{-,r})^N \eta$,
- if $\pi = 1$, we select $s \in [0..r-1]$ uniformly at random, and set $\mu[a_j..b_j] = (\tau_{+,r,s})^N \eta$.

Finally, the initial and final fragments μ_i and μ_f of μ are chosen to be the shortest words that label a transition from q_0 to q_* and q_* to a final state, respectively.

8.3.3.3 Properties of the distribution $\mathcal{D}$

Next, we establish that the distribution $\mathcal{D}$ has the desired properties.

Observation 8.3.19. *If ε is small enough, $\mathcal{D}$ is well-defined, i.e. for every t between 0 and $\log((S\varepsilon)^{-1})$, we have $0 \le p_t \le 1$.*

Observation 8.3.20. *If $\pi = 1$, then $\mu \in \mathcal{PL}(\mathcal{A})$.*

Lemma 8.3.21. *Conditioned on $\pi = 0$, the probability of the event $\mathcal{F} = \{\mu$ is ε-far from $\mathcal{PL}(\mathcal{A})\}$ goes to 1 as n goes to infinity.*

Proof. When $\pi = 0$, the procedure for sampling μ puts blocking factors of the form $(i_* : x)$ at positions equal to $i_* \mod \lambda$. Any word containing such a factor at such a position is not in $\mathcal{PL}(\mathcal{A})$, therefore any sequence of substitutions that transforms μ into a word of $\mathcal{PL}(\mathcal{A})$ must make at least one substitution in every such factor. Consequently, the

distance between μ and $\mathcal{PL}(\mathcal{A})$ is at least the number of blocking factors in μ. To prove the lemma, we show that this number is at least εn with high probability, by showing that it is larger than εn by a constant factor in expectation and using a concentration argument.

Let B_j denote the number of blocking factors in the j-th interval: it is equal to $2^{-\kappa_j}/(S\varepsilon)$ when $\kappa_j > 0$ and to 0 otherwise.

▷ Claim 8.3.22. Let $B = \sum_{j=1}^{k} B_j$, and let $E = \mathbb{E}[B]$. We have $E \geq 2\varepsilon n$.

Claim proof. By direct calculation:

$$\begin{aligned} E &= \sum_{j=1}^{k} \mathbb{E}[B_j] = \sum_{j=1}^{k} \sum_{t=1}^{\log(S/\varepsilon)} 2^{-t}/(S\varepsilon) \cdot p_t \\ &= \sum_{j=1}^{k} \sum_{t=1}^{\log(S/\varepsilon)} 2^{-t}/(S\varepsilon) \cdot 3 \cdot 2^t \varepsilon S / \log(S/\varepsilon) = \sum_{j=1}^{k} \sum_{t=1}^{\log(S/\varepsilon)} 3/\log(S/\varepsilon) \\ &= 3k \geq 2\varepsilon n \end{aligned}$$

□

We will now show that $\mathbb{P}(B < \varepsilon n)$ goes to 0 as n goes to infinity. We use Hoeffding's inequality, which we recall here:

Fact 8.3.23 ([174, Theorem 2])**.** *Let $X_1, \ldots, X_k$ be independent random variables such that for every $i = 1, \ldots, k$, we have $a_i \leq X_i \leq b_i$, and let $S = \sum_{i=1}^{k} X_i$. Then, for any $t > 0$, we have*

$$\mathbb{P}(\mathbb{E}[S] - S \geq t) \leq \exp\left(-\frac{2t^2}{\sum_{i=1}^{k}(b_i - a_i)^2}\right).$$

By Claim 8.3.22, we have $B < \varepsilon n \Rightarrow E - B \geq \varepsilon n$, and therefore $\mathbb{P}(B < \varepsilon n) \leq \mathbb{P}(E - B \geq \varepsilon n)$. The random variable B is the sum of k independent random variables, each taking values between 0 and $1/(S\varepsilon)$. Therefore, by Hoeffding's inequality (Fact 8.3.23), we have

$$\begin{aligned} \mathbb{P}(E - B < \varepsilon n) &\leq \exp\left(-\frac{2\varepsilon^2 n^2}{k/(S\varepsilon)^2}\right) \\ &\leq \exp\left(-\frac{2S^2\varepsilon^4 n^2}{\varepsilon n}\right) \text{ as } k \leq \varepsilon n \\ &\leq \exp\left(-2S^2\varepsilon^3 n\right) \end{aligned}$$

This probability goes to 0 as n goes to infinity, which concludes the proof. □

Corollary 8.3.24. *For large enough n, we have $\mathbb{P}(\mathcal{F}) \geq 5/12$.*

Intuitively, our distribution is hard to test because positive and negative instances are very similar. Therefore, a tester with few queries will likely not be able to tell them apart: the perfect completeness constraint forces the tester to accept in that case. Below, we establish this result formally.

Lemma 8.3.25. *Let T be a deterministic tester with perfect completeness (i.e. it always accepts $\tau \in \mathcal{PL}(\mathcal{A})$) and let q_j denote the number of queries that it makes in the j-th interval. Conditioned on the event $\mathcal{M} = \{\forall j \text{ s.t. } \kappa_j > 0, q_j < 2^{\kappa_j}\}$, the probability that T accepts $\mu \sim \mathcal{D}$ is 1.*

Proof. We proceed by contradiction, and show that if there exists a word τ with non-zero probability w.r.t. $\mathcal{D}$ under $\mathcal{M}$ that T rejects, then there exists a word $\tau' \in \mathcal{PL}(\mathcal{A})$ that T rejects that also has non-zero probability, contradicting the fact that T has perfect completeness.

Let τ be the word rejected by T: as T has perfect completeness, τ is not in $\mathcal{PL}(\mathcal{A})$, and there must be at least one interval with $\kappa_j > 0$. Consider every interval j such that $\kappa_j > 0$: it is of the form $(\tau_{-,r})^N \eta$ where $r = 2^{\kappa_j}$ and $\tau_{-,r} = \phi(\nu_-)^r \chi$. Therefore, if $q_j < 2^{\kappa_j}$, then there is a copy of ν_- that has not been queried by T across all copies of $\tau_{-,r}$. Consider the word τ' obtained by replacing this copy of ν_- with ν_+ in all N copies of $\tau_{-,r}$ in the block. The result block is of the form $(\tau_{+,r,s})^N \eta$ for some $s < r$, and by construction it is not blocking. Applying this operation to all blocks results in a word τ' that is in $\mathcal{PL}(\mathcal{A})$. Furthermore, τ' has non-zero probability under $\mathcal{D}$ conditioned on $\mathcal{M}$: it can be obtained by flipping the random bit π and choosing the right index s in every block. □

Next, we show that if a tester makes few queries, then the event $\mathcal{M}$ has large probability.

Lemma 8.3.26. *Let T be a deterministic tester, let q_j denote the number of queries that it makes in the j-th interval, and assume that T makes at most $\frac{1}{72} \cdot \log(S/\varepsilon)/\varepsilon$ queries, i.e. $\sum_j q_j \leq \frac{1}{72} \cdot \log(S/\varepsilon)/\varepsilon$. The probability of the event $\mathcal{M} = \{\forall j \text{ s.t. } \kappa_j > 0, q_j < 2^{\kappa_j}\}$ is at least $11/12$.*

Proof. We show that the probability of $\overline{\mathcal{M}}$, the complement of $\mathcal{M}$, is at most $1/12$. We have:

$$\begin{aligned}
\mathbb{P}\left(\overline{\mathcal{M}}\right) &= \mathbb{P}\left(\exists j : \kappa_j > 0 \wedge q_j \geq 2^{\kappa_j}\right) \\
&\leq \sum_j \mathbb{P}\left(\kappa_j > 0 \wedge q_j \geq 2^{\kappa_j}\right) && \text{by union bound} \\
&\leq \sum_j \sum_{t=1}^{\lfloor \log q_j \rfloor} p_t = \sum_j \sum_{t=1}^{\lfloor \log q_j \rfloor} \frac{3 \cdot 2^t \varepsilon}{\log(S/\varepsilon)} && \text{by def. of } p_t \\
&\leq \frac{3\varepsilon}{\log(S/\varepsilon)} \sum_j \sum_{t=1}^{\lfloor \log q_j \rfloor} 2^t
\end{aligned}$$

By upper bounding the sum of power of 2 up to $k = \lfloor \log q_j \rfloor$ by 2^{k+1}, we obtain:

$$\begin{aligned}
\mathbb{P}\left(\overline{\mathcal{M}}\right) &\leq \frac{3\varepsilon}{\log(S/\varepsilon)} \sum_j 2q_j \\
&= \frac{3\varepsilon}{\log(S/\varepsilon)} \cdot \frac{2}{72} \cdot \frac{\log(S/\varepsilon)}{\varepsilon} \\
&\leq 1/12
\end{aligned}$$

□

We are now ready to prove Theorem 8.3.14.

Proof of Theorem 8.3.14. We want to show that any tester with perfect completeness for $\mathcal{L}(\mathcal{A})$ requires at least $\frac{1}{72} \cdot \log(S/\varepsilon)/\varepsilon$ queries, by showing that any tester with fewer queries errs with probability at least $1/3$. We show that any **deterministic** algorithm T with perfect completeness that makes less than $\frac{1}{72} \cdot \log(S/\varepsilon)/\varepsilon$ queries errs on u when $(0 : u) \sim \mathcal{D}$ with probability at least $1/3$, and conclude using Yao's Minimax principle.

Consider such an algorithm T. The probability that T makes an error on u is lower-bounded by the probability that u is ε-far from $\mathcal{L}(\mathcal{A})$ and T accepts, which in turn is larger than the probability of $\mathcal{M} \cap \mathcal{F}$. By Corollary 8.3.24, we have $\mathbb{P}(\mathcal{F}) \geq 5/12$, and by Lemma 8.3.26, $\mathbb{P}(\mathcal{M})$ is at least $11/12$. Therefore, we have

$$\mathbb{P}(T \text{ errs}) \geq \mathbb{P}(\mathcal{M} \cap \mathcal{F}) \geq 1 - 7/12 - 1/12 = 4/12 = 1/3.$$

This concludes the proof of Theorem 8.3.14, and consequently of Theorem 8.3.2. □

8.4 Characterisation of Hard Languages for All NFAs

In this section we extend the results of the previous section to all finite automata. This extension is based on a generalization of blocking factors: we introduce *blocking sequences*, which are sequences of factors that witness the fact that we cannot take any path through the strongly connected components of the automaton. For the lower bound, we define a suitable partial order on blocking sequences, which extends the factor relation on words to those sequences, and allows us to define *minimal* blocking sequences.

8.4.1 Blocking sequences

8.4.1.1 Examples motivating blocking sequences

Before presenting the technical part of the proof, let us go through two examples, which motivate the notions that we introduce and illustrate some of the main difficulties.

Example 8.4.1. Consider the automaton $\mathcal{A}_1$ depicted in Fig. 8.4: it recognizes the language L_1 of words in which all c's appear before the first b, over the alphabet $\{a, b, c\}$.

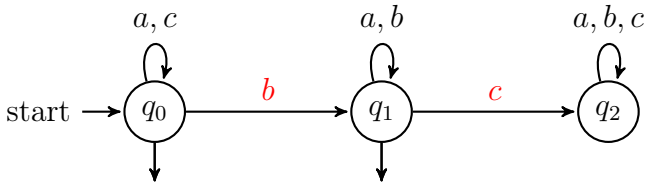

Figure 8.4: An automaton $\mathcal{A}_1$ that recognizes the language $L_1 = (a + c)^*(a + b)^*$.

The set of minimal blocking factors of $\mathcal{A}_1$ is infinite: it is the language ba^*c. Yet, L_1 is easy to test: we sample $O(1/\varepsilon)$ letters at random, answer "no" if the sample contains a c occurring after a b, and "yes" otherwise. To prove that this yields a property tester, we rely on the following property:

Property 8.4.2. *If u is ε-far from L_1, then it can be decomposed into $u = u_1u_2$ where u_1 contains $\Omega(\varepsilon n)$ letters b and u_2 contains $\Omega(\varepsilon n)$ letters c.*

The pair of factors (b, c) is an example of blocking sequence: a word that contains an occurrence of the first followed by an occurrence of the second cannot be in L_1. We can also show that a word ε-far from L_1 must contains many disjoint blocking sequences – this property (Lemma 8.4.18) underpins the algorithm for general regular languages.

What this example shows is that blocking factors are not enough: we need to consider sequences of factors, yielding the notion of *blocking sequences*. Intuitively, a blocking

sequence for L is a sequence $\sigma = (v_1, \ldots, v_k)$ of (positional) words such that if each word of the sequence appears in u, with the occurrences of the v_i's ordered as in σ, then u is not in L.[5] While L_1 has infinitely many minimal blocking factors, it has a single minimal blocking sequence $\sigma = (b, c)$.

Notice that the (unique) blocking sequence (b, c) can be visualized on Fig. 8.4: it is composed of the red letters that label transitions between the different SCCs. This is no coincidence: in many simple cases, blocking sequences are exactly sequences that contain one blocking factors for each SCC. This fact could lead one to believe that the set of minimal blocking sequences is exactly the set of sequences of minimal blocking factors, one for each SCC. In particular, this would imply that as soon as one SCC has infinitely many minimal blocking factors, the language of the whole automaton is hard to test. We show in the next example that this is not always the case, because SCCs might share minimal blocking factors.

Example 8.4.3. Consider the automaton in Fig. 8.5: it has two SCCs and a sink state. The minimal blocking factors of the first SCC are given by $B_1 = be^*c + a$, and $B_2 = \{a\}$ for the second SCC. This automaton is easy to test: intuitively, a word that is ε-far from this language has to contain many a's, as otherwise we can make it accepted by deleting all a's, thanks to the second SCC. However, a is also a blocking factor of the first SCC, therefore, as soon as we find two a's in the word, we know that it is not in L_2.

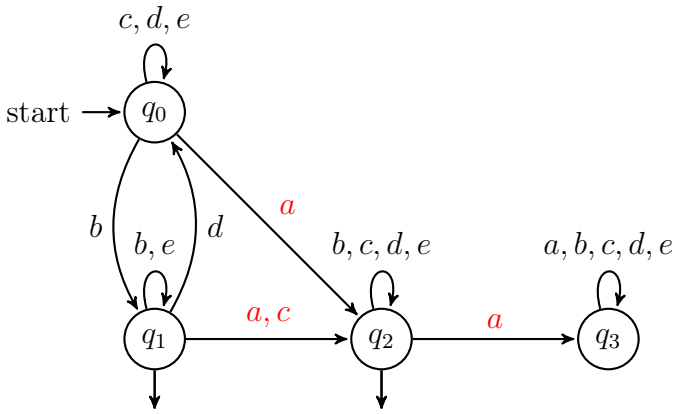

Figure 8.5: An automaton $\mathcal{A}_2$ that recognizes the language $L_2 = [((c + d + e)^*b(b + e)^*d)^*a](b + c + d + e)^*$.

The crucial facts here are that the set B_2 of minimal blocking factors of the second SCC is finite and it is a subset of B_1: the infinite nature of B_1 is made irrelevant because any word far from the language contains many a's. Therefore, $\mathcal{A}_2$ has a *single* minimal blocking sequence, $\sigma = (a)$.

8.4.1.2 Portals and SCC-paths

Intuitively, blocking sequences are sequences of blocking factors of successive strongly connected components. To formalize this intuition, we use *portals*, which describe how a run in the automaton interacts with a strongly connected component, and *SCC-paths*, that describe a succession of portals.

[5] This is not quite the definition, but it conveys the right intuition.

In what follows, we fix an NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, \{q_f\})$. We assume w.l.o.g. that $\mathcal{A}$ has a unique final state q_f. Let $\mathscr{S}$ be the set of SCCs of $\mathcal{A}$. We define the partial order relation $\leq_{\mathcal{A}}$ on $\mathscr{S}$ by $S \leq_{\mathcal{A}} T$ if and only if T is reachable from S. We write $<_{\mathcal{A}}$ for its strict part $\leq_{\mathcal{A}} \setminus \geq_{\mathcal{A}}$. These relations can be naturally extended to states through their SCC: if $s \in S$ and $t \in T$, then $s \leq_{\mathcal{A}} t$ if and only if $S \leq_{\mathcal{A}} T$.

We define p as the least common multiple of the lengths of all simple cycles of $\mathcal{A}$. Given a number $k \in \mathbb{Z}/p\mathbb{Z}$, we say that a state t is k-reachable from a state s if there is a path from s to t of length k modulo p. In what follows, we use "positional words" for p-positional words with this value of p.

Remark 8.4.4. In the rest of this section we will not try to optimize the constants in the formulas. They will, in fact, become quite large in some of the proofs. We make this choice to make the proofs more readable, although some of them are already technical.

For instance, the choice of p as the lcm of the lengths of simple cycles is not optimal: we could use, for instance, the lcm of the periods of the SCCs.

Definition 8.4.5 (Portal)**.** *A* portal *is a 4-tuple $P = s, x \rightsquigarrow t, y \in (Q \times \mathbb{Z}/p\mathbb{Z})^2$, such that s and t are in the same SCC. It describes the first and last states visited by a path in an SCC, and the positions x, y (modulo p) at which it first and lasts visits that SCC.*

The positional language of a portal is the set

$$\mathcal{L}(s, x \rightsquigarrow t, y) = \{(x : w) \mid s \xrightarrow{w} t \wedge x + |w| = y \pmod p\}.$$

Portals were already defined by Alon et al. [24], in a slightly different way. Our definition will allow us to express blocking sequences more naturally.

Definition 8.4.6. *A positional word $(n : u)$ is blocking for a portal P if it is not a factor of any word of $\mathcal{L}(P)$. In other words, there is no path that starts in s and ends in t, of length $y - x$ modulo p, which reads u after $n - x$ steps modulo p.*

The above definition matches the definition of blocking factors for strongly connected automata. This is no coincidence: we show in the next lemma that the language of a portal has a strongly connected automaton.

Lemma 8.4.7. *Let $\mathcal{A}$ be an automaton and P a portal of $\mathcal{A}$. There is a strongly connected NFA with at most $p|\mathcal{A}|$ states that recognizes $L' = \mathcal{L}(P)$.*

Proof. Let S denote the SCC of s and t in $\mathcal{A}$, and let λ denote its period. By definition of p, λ divides p: let k be the integer such that $p = \lambda k$. The automaton $\mathcal{A}'$ for L' simulates the behavior of $\mathcal{A}$ restricted to S starting from the state s, while keeping track of the number of letters read modulo p, starting from x. More precisely, let $Q_0, \dots Q_{\lambda-1}$ be the partition of the states of S given by Fact 8.2.4. The set of states of $\mathcal{A}'$ is given by

$$Q' = \{(s', i + j\lambda) \mid s' \in Q_i \wedge i = 0 \dots, \lambda - 1 \wedge j = 0, \dots, k - 1\}.$$

It is a subset of $Q \times \mathbb{Z}/p\mathbb{Z}$, hence it has cardinality at most $p|\mathcal{A}|$. The transitions in $\mathcal{A}'$ are of the form $(s_1, i + j\lambda) \xrightarrow{(i,a)} (s_2, i + j\lambda + 1 \pmod p)$ for any s_1, s_2 such that $s_1 \xrightarrow{a} s_2$ in $\mathcal{A}$.

Furthermore, $\mathcal{A}'$ is strongly connected. Let i_1, i_2 be indices of periodicity classes of S, and let $s_1 \in Q_{i_1}, s_2 \in Q_{i_2}$ be states of S. We show that for any $j_1, j_2 < k$, there is a path from $\sigma_1 = (s_1, i_1 + j_1\lambda)$ to $\sigma_2 = (s_2, i_2 + j_2\lambda)$ in $\mathcal{A}'$. Let ℓ be a sufficiently large integer

equal to $(i_2 - i_1) + (j_2 - j_1)\lambda \pmod p$. As λ divides p, ℓ is equal to $(i_2 - i_1) \pmod \lambda$. By taking ℓ larger than the reachability constant of S, Fact 8.2.4 gives us that there is a path of length ℓ from s_1 to s_2 in S, labeled by some word u. The positional word $(x : u)$ labels a transition from σ_1 to σ_2 in $\mathcal{A}'$, hence it is strongly connected.

Note that the period of $\mathcal{A}'$ is p, hence we can apply the results we obtained on strongly connected NFAs in Section 8.3 to portals, with $p|\mathcal{A}|$ as the number of states and p as the period. □

Portals describe the behavior of a run inside a single strongly connected component of the automaton. Next, we introduce SCC-paths, which describe the interaction of a run with multiple SCCs and between two successive SCCs.

Definition 8.4.8 (SCC-path)**.** *An* SCC-path π *of* $\mathcal{A}$ *is a sequence of portals linked by single-letter transitions* $\pi = s_0, x_0 \rightsquigarrow t_0, y_0 \xrightarrow{a_1} s_1, x_1 \rightsquigarrow t_1, y_1 \cdots \xrightarrow{a_k} s_k, x_k \rightsquigarrow t_k, y_k$, *such that for all* $i \in \{1, \dots, k\}$, $x_i = y_{i-1} + 1 \pmod p$, $t_{i-1} \xrightarrow{a_i} s_i$, *and* $t_{i-1} <_{\mathcal{A}} s_i$.

Intuitively, an SCC-path is a description of the states and positions at which a path through the automaton enters and leaves each SCC.

Definition 8.4.9. *The language* $\mathcal{L}(\pi)$ *of an SCC-path* $\pi = P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots P_k$ *is the set*

$$\mathcal{L}(\pi) = \mathcal{L}(P_0)a_1\mathcal{L}(P_2)a_2 \cdots \mathcal{L}(P_k).$$

We say that π is *accepting* if $P_0 = s_0, x_0 \rightsquigarrow t_0, y_0$, $P_k = s_k, x_k \rightsquigarrow t_k, y_k$ with $x_0 = 0$, $s_0 = q_0$, $t_k = q_f$ and $\mathcal{L}(\pi)$ is non-empty.

Lemma 8.4.10. *We have* $\mathcal{PL}(\mathcal{A}) = \bigcup_{\pi \text{ accepting}} \mathcal{L}(\pi)$.

Proof. We show that for any word μ in $\mathcal{PL}(\mathcal{A})$, there is an accepting SCC-path π whose language contains μ. Let $\mu = a_1 \cdots a_n$ be a word of length n in $\mathcal{PL}(\mathcal{A})$: there exists an accepting run $\rho = q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_n} q_n = q_f$ in $\mathcal{A}$.

We define the sequence of indices $i_0 < i_1 < \ldots < i_k < i_{k+1}$ as follows:

- $i_0 = 0, i_{k+1} = n + 1$,
- for every $j = 1, \dots, k$, i_j is the smallest index such that $q_{i_j - 1} <_{\mathcal{A}} q_{i_j}$, i.e. $q_{i_j - 1}$ and q_{i_j} belong to distinct SCCs.

In other words, those are the indices at which ρ enters a new SCC. We then define the SCC-path $\pi(\rho)$ as follows:

$$\pi(\rho) = q_0, 0 \rightsquigarrow q_{i_1 - 1}, y_0 \xrightarrow{a_{i_1}} q_{i_1}, x_1 \rightsquigarrow q_{i_2 - 1}, y_1 \cdots \xrightarrow{a_{i_k}} q_{i_k}, x_k \rightsquigarrow q_n, y_k$$

where $x_j = i_j \pmod p$ and $y_j = x_{j+1} - 1 \pmod p$ for all $j = 0, \dots, k + 1$.

By construction, $\mu \in \mathcal{L}(\pi(\rho))$ and $\pi(\rho)$ is an accepting SCC-path.

The converse inclusion follows by definition of (accepting) SCC-paths. □

As a consequence the distance between a word μ and the (positional) language of $\mathcal{A}$ is equal to the minimum of the distances between μ and the languages of the SCC-paths of $\mathcal{A}$.

Corollary 8.4.11. *For any positional word* μ, *we have*

$$d(\mu, \mathcal{PL}(\mathcal{A})) = \min_{\pi \text{ accepting}} d(\mu, \mathcal{L}(\pi)).$$

Decomposing $\mathcal{A}$ as a union of SCC-paths allows us to use them as an intermediate step to define blocking sequences. We earlier defined blocking factors for portals: we now generalize this definition to blocking sequences for SCC-paths, to finally define blocking sequence of automata.

Definition 8.4.12 ((Strongly) Blocking Sequences for SCC-paths)**.** *We say that a sequence $\sigma = (\mu_1, \dots, \mu_\ell)$ of positional factors is blocking for an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ if there is a sequence of indices $i_0 \leq i_1 \leq \cdots \leq i_k$ such that for every j, μ_{i_j} is blocking for P_j.*

Furthermore, if there is a sequence of indices $i_0 < i_1 < \cdots < i_k$ with the same property, then σ is said to be strongly blocking *for π.*

Note that, crucially, in the definition of blocking sequences, consecutive indices i_j and i_{j+1} can be equal, i.e. a single factor of the sequence may be blocking for multiple consecutive SCCs in the SCC-path. This choice is motivated by Example 8.4.3, where the language is easy because consecutive SCCs share blocking factors.

We say that two occurrences of blocking sequences in a word μ are *disjoint* if the occurrences of their factors appear on disjoint sets of positions in μ.

In the strongly connected case, we had the property that if μ contains an occurrence of a factor blocking for $\mathcal{A}$, then μ is not in the language of $\mathcal{A}$. The following lemma gives an extension of this result to *strongly* blocking sequences and the language of an SCC-path.

Lemma 8.4.13. *Let π be an SCC-path. If μ contains a strongly blocking sequence for π, then $\mu \notin \mathcal{L}(\pi)$.*

Proof. We proceed by induction on the length k of the SCC-path $\pi = P_0 \xrightarrow{a_1} P_1 \cdots \xrightarrow{a_k} P_k$. Let $\sigma = (\nu_0, \dots, \nu_k)$ be a strongly blocking sequence for π that occurs in μ. If $k = 0$, then σ consists only of a blocking factor for P_0, hence μ is not in $\mathcal{L}(P_0)$, which is equal to $\mathcal{L}(\pi)$.

For $k > 0$, assume for the sake of contradiction that $\mu \in \mathcal{L}(\pi)$. By definition of $\mathcal{L}(\pi)$, μ can then be written as $\mu_0 a_1 \mu'$, with $\mu_0 \in \mathcal{L}(P_0)$ and $\mu' \in \mathcal{L}(\pi')$, where $\pi' = P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} P_k$. As ν_0 is blocking for P_0, the prefix μ_0 of μ must end before the occurrence of ν_0 in μ, and the sequence $\sigma' = (\nu_1, \dots, \nu_k)$ occurs in μ'. Furthermore, because σ is *strongly* blocking for π, σ' is strongly blocking for π'. Using the induction hypothesis on μ' and the path π' of length $k-1$, this implies that $\mu' \notin \mathcal{L}(\pi')$, a contradiction. □

We can now define sequences that are blocking for an automaton: they are sequences that are blocking for *every* accepting SCC-path of the automaton.

Definition 8.4.14 (Blocking sequence for $\mathcal{A}$)**.** *Let $\sigma = (\mu_1, \dots, \mu_\ell)$ be a sequence of positional words. We say that σ is blocking for $\mathcal{A}$ if it is blocking for all accepting SCC-paths of $\mathcal{A}$.*

As an example, observe that the sequences $((0 : ab), (1 : ab))$ and $((0 : aa), (0 : b))$ are both blocking for the automaton displayed in Fig. 8.6 (see Example 8.4.15).

Example 8.4.15. Consider the automaton displayed in Fig. 8.6. The lcm of the lengths of its simple cycles is $p = 2$. This automaton has six accepting SCC-paths, including

$$\pi_1 = q_0, 0 \rightsquigarrow q_0, 0 \xrightarrow{a} q_1, 1 \rightsquigarrow q_1, 1 \xrightarrow{a} q_3, 0 \rightsquigarrow q_3, 0 \xrightarrow{b} q_4, 1 \rightsquigarrow q_4, 1$$

$$\pi_2 = q_0, 0 \rightsquigarrow q_0, 0 \xrightarrow{a} q_2, 1 \rightsquigarrow q_1, 0 \xrightarrow{a} q_3, 1 \rightsquigarrow q_3, 0 \xrightarrow{b} q_4, 1 \rightsquigarrow q_4, 1$$

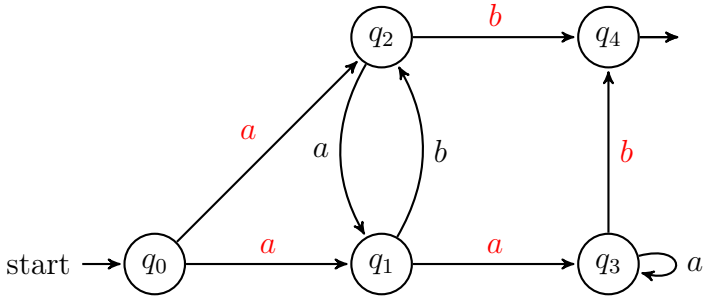

Figure 8.6: Automaton used for Example 8.4.15.

The language of the portal π_1 is $a(ba)^*a(a^2)^*b$. A blocking sequence for this SCC-path is $((0:aa),(0:b))$, which is in fact blocking for all of the SCC-paths.

On the other hand, $((0:ab))$ is not blocking for π_1, as $(0:ab)$ is not a blocking factor for the portal $q_1, 1 \rightsquigarrow q_1, 1$. It is, however, a blocking sequence for π_2. This is because if we enter the SCC $\{q_1, q_2\}$ through q_1, a factor ab can only appear after an even number of steps, while if we enter through q_2, it can only appear after an odd number of steps.

8.4.2 An efficient property tester

In this section, we show that for any regular language L and any small enough $\varepsilon > 0$, there is an ε-property tester for L that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.

Theorem 8.4.16. *For any NFA $\mathcal{A}$ and any small enough $\varepsilon > 0$, there exists an ε-property tester for $\mathcal{L}(\mathcal{A})$ that uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries.*

As mentioned in the overview, this result supersedes the one that we obtained in [48]: while both testers use the same number of queries, the tester in [48] works under the edit distance, while that of Theorem 8.4.16 is designed for the Hamming distance. As the edit distance never exceeds the Hamming distance, the set of words that are ε-far with respect to the former is contained in the set of words ε-far for the latter. Therefore, an ε-tester for the Hamming distance is also an ε-tester for the edit distance, and this result is stronger.

The property tester behind Theorem 8.4.16 uses the property tester for strongly connected NFAs as a subroutine, and its correctness is based on an extension of Lemma 8.3.13 to blocking sequences. We show that we can reduce property testing of $\mathcal{L}(\mathcal{A})$ to a search for blocking sequences in the word, in the following sense:

- If μ contains a strongly blocking sequence for each of the SCC-paths of $\mathcal{A}$, then it is not in the language and we can answer no (Corollary 8.4.17).
- If μ is ε-far from the language, then for each accepting SCC-path π of $\mathcal{A}$, μ is far from for the language of π and contains many disjoint strongly blocking sequences for π (Lemma 8.4.18), hence random sampling is likely to find at least one of them, and we reject μ with constant probability.

Corollary 8.4.17. *If μ contains a strongly blocking sequence for each SCC-path of $\mathcal{A}$, then $\mu \notin \mathcal{PL}(\mathcal{A})$.*

Proof. This follows from Lemma 8.4.10. □

The next lemma expresses a partial converse to Corollary 8.4.17 and generalizes Lemma 8.3.12 from the strongly connected case: if a word is far from the language, then it contains many strongly blocking sequences for any SCC-path.

Lemma 8.4.18. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path, let $L = \mathcal{L}(\pi)$, and let μ be a positional word of length n such that $d(\mu, L)$ is finite. There is a constant C such that if $n \geq C/\varepsilon$ and μ is ε-far from L, then μ can be partitioned into $\mu = \mu_0\mu_1\cdots\mu_k$ such that for every $i = 0, \ldots, k$, μ_i contains at least $\frac{\varepsilon n}{C}$ disjoint blocking factors for P_i, each of length at most $O(1/\varepsilon)$.*

Proof. We proceed similarly to the proof of Lemma 8.3.12, and only sketch this proof. Starting from the left end of μ, we accumulate letters until we find a factor blocking for P_0, and iterate again starting from p positions later, where p is the lcm of the length of all cycles in $\mathcal{A}$; notably, it is a multiple of the reachability constant of a strongly connected automaton recognizing $\mathcal{L}(P_0)$. When we have found at least $K = \frac{\varepsilon n}{C}$ blocking factors (C is to be determined later) for $\mathcal{L}(P_i)$, this position marks the end of μ_i, and we iterate with the next portal in π.

Let us assume that the process ends (i.e. we reach the right end of μ) before finding enough blocking factors for all portals. We show that in this case, the distance between μ and L is at most εn. Assume that we stop before finding enough blocking factors for the i-th portal, P_i. As in the proof of Lemma 8.3.12, we replace the last letter of each blocking factor and use the padding between them to make the run accepted by the SCC-path: this uses at most $((i+1)\cdot(K+1)+2)p$ substitutions. If we set $C = 4(k+3)p$, this is less than εn when $n \geq C/\varepsilon$. Therefore, if μ is ε-far from $\mathcal{L}(\pi)$, then the decomposition process finds at least K blocking factors for P_i in μ_i for each i.

Then, since all of these factors are disjoint, we can use the same technique as in Lemma 8.3.13 to show that at least half of these factors have length $O(1/\varepsilon)$, and the result holds, up to doubling C. □

Corollary 8.4.19. *Let $L = \mathcal{PL}(\mathcal{A})$ and let μ be a positional word of length n. If L contains a word of length n and μ is ε-far from L, then μ contains $\Omega(\varepsilon n)$ disjoint blocking sequences for $\mathcal{A}$.*

Proof. We use a proof identical to that of Lemma 8.4.18, except that we consider a linear ordering of all the portals of $\mathcal{A}$ given by topological ordering, instead of the linear given by an SCC-path. The graph used for the topological ordering is the graph of all portals of $\mathcal{A}$, with an edge from P to P' when P and P' appear consecutively in some SCC-path of $\mathcal{A}$. Since any two portals in an SCC-path are from different SCCs of $\mathcal{A}$, this graph is acyclic, and its vertices can be topologically ordered. □

We are now ready to prove Theorem 8.4.16.

Proof of Theorem 8.4.16. Our algorithm iterates over all K accepting SCC-paths $\pi = P_0 \xrightarrow{a_1} \ldots \xrightarrow{a_k} P_k$ of $\mathcal{A}$, and for each π, searches for blocking sequences for π in $\mu = (0 : u)$. If we find a strongly blocking sequence for π in μ, then by Lemma 8.4.13, μ is not in $\mathcal{PL}(\mathcal{A})$ and we can reject. Note that if $\mu \in \mathcal{PL}(\mathcal{A})$, then the algorithm will not reject, hence the perfect completeness property is satisfied.

Next, we show that if μ is ε-far from $\mathcal{PL}(\mathcal{A})$, then we can find a strongly blocking sequence for π with probability at least $1 - 1/(3K)$ using $O(\log(\varepsilon^{-1})/\varepsilon)$ queries. Our algorithm is based on the following observation:

Observation 8.4.20. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path. Let $\nu_0, \ldots, \nu_k$ be positional words such that ν_i is blocking for P_i. Then, $\sigma = (\nu_0, \ldots, \nu_k)$ is a strongly blocking sequence of π.*

We can assume w.l.o.g. that μ has length at least $2C/\varepsilon$, where C is the constant defined in Lemma 8.4.18, otherwise we can read all of μ using $O(1/\varepsilon)$ queries. Therefore, we can apply Lemma 8.4.18, and μ can be partitioned into $k+1$ words $\mu_0, \ldots, \mu_k$ such that each μ_i contains at least $\varepsilon n/C$ disjoint blocking factors for C, each of length $L = O(1/\varepsilon)$.

For each i, we can use the algorithm of Lemma 8.3.9 to sample from μ a set $\mathcal{F}$ that contains a factor that contains a ν_i with probability at least $2/3$. By repeating the procedure $O(\ln(3K \cdot (k+1)))$ times and taking the union of all returned sets $\mathcal{F}$, we can increase this probability to $1 - \frac{1}{3K \cdot (k+1)}$. Then, by the union bound, we find a blocking factor ν_i for each P_i in the corresponding μ_i with probability at least $1 - 1/(3K)$. As observed above, the sequence $\sigma = (\nu_0, \ldots, \nu_k)$ is strongly blocking for π.

By union bound again, this algorithm finds a strongly blocking sequence for each of the K SCC-paths in $\mathcal{A}$, and therefore rejects μ, with probability at least $2/3$.

For a single μ_i of a given SCC-path, the sampling procedure uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries (by Claim 8.3.8). As the lengths and number of SCC-paths in $\mathcal{A}$ does not depend on the input length, this algorithm uses $O(\log(\varepsilon^{-1})/\varepsilon)$ queries in total. □

8.4.3 Lower bound

In order to characterize hard languages for all automata, we define a partial order $\trianglelefteq$ on sequences of positional factors. It is an extension of the factor partial order on blocking factors. It will let us define *minimal blocking sequences*, which we use to characterize the complexity of testing a language.

Definition 8.4.21 (Minimal blocking sequence)**.** *Let $\sigma = (\mu_1, \mu_2, \ldots, \mu_k)$ and $\sigma' = (\mu'_1, \ldots, \mu'_t)$ be sequences of positional words. We have $\sigma \trianglelefteq \sigma'$ if there exists a sequence of indices $i_1 \leq i_2 \leq \ldots \leq i_k$ such that μ_j is a factor of μ'_{i_j} for all $j = 1, \ldots, k$.*

A blocking sequence σ of $\mathcal{A}$ (resp. π) is minimal *if it is a minimal element of $\trianglelefteq$ among blocking sequences of $\mathcal{A}$ (resp. π). The set of minimal blocking sequences of $\mathcal{A}$ (resp. π) is written $\mathsf{MBS}(\mathcal{A})$ (resp. $\mathsf{MBS}(\pi)$).*

Remark 8.4.22. If $\sigma \trianglelefteq \sigma'$ and σ is a blocking sequence for an SCC-path π then σ' is also a blocking sequence for π.

We make the remark that minimal blocking sequences have a bounded number of terms. This is because if we build the sequence from left to right by adding terms one by one, the minimality implies that at each step we should block a previously unblocked portal.

Lemma 8.4.23. *A minimal blocking sequence for $\mathcal{A}$ contains at most $p^2|Q|^2$ terms.*

Proof. First, remark that there at most $p^2|Q|^2$ portals in $\mathcal{A}$. Let $\sigma = (\mu_1, \ldots, \mu_\ell)$ be a minimal blocking sequence for $\mathcal{A}$. For all $i = 1, \ldots, \ell$, we define $\sigma_i = (\mu_1, \ldots, \mu_i)$, and σ_0 is the empty sequence.

Then, for each i, we consider the set $\mathcal{S}_i$ of portals P such that for all accepting SCC-path π of $\mathcal{A}$ containing P, the prefix of π ending at P is blocked by σ_i. We have $\mathcal{S}_0 = \emptyset$, and $\mathcal{S}_\ell$ is the set of all portals of $\mathcal{A}$.

We claim that for every $i < \ell$, $\mathcal{S}_i$ is a proper subset of $\mathcal{S}_{i+1}$. Otherwise, if $\mathcal{S}_i = \mathcal{S}_{i+1}$, then removing μ_{i+1} from σ gives a blocking sequence σ' of $\mathcal{A}$, such that $\sigma' \trianglelefteq \sigma$, contradicting the minimality of σ. Therefore, it follows that $\ell \leq p^2|Q|^2$. □

8.4.3.1 Reducing to the strongly connected case

To prove a lower bound on the number of queries necessary to test a language when $\mathsf{MBS}(\mathcal{A})$ is infinite, we present a reduction to the strongly connected case. Under the assumption that $\mathcal{A}$ has infinitely many minimal blocking sequences, we exhibit a portal P of $\mathcal{A}$ with infinitely many minimal blocking factors and "isolate it" by constructing two sequences of positional factors σ_l and σ_r such that for all μ, $\sigma_l, (\mu), \sigma_r$ is blocking for $\mathcal{A}$ if and only if μ is a blocking factor of P. Then we reduce the problem of testing the language of this portal to the problem of testing the language of P.

To define "isolating P" formally, we define the left (and right) effect of a sequence on an SCC-path. Informally, the left effect of a sequence σ on an SCC-path π is related to the index of the first portal in π where a run can be after reading σ, because all previous portals have been blocked. The right effect represents the same in reverse, starting from the end of the run.

More formally, the *left effect* of a sequence σ on an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ is the largest index i such that the sequence is blocking for $P_0 \xrightarrow{a_1} \cdots P_i$ (-1 if there is no such i). We denote it by $[\sigma \gg \pi]$. Similarly, the *right effect* of a sequence on π is the smallest index i such that the sequence is blocking for $P_i \xrightarrow{a_{i+1}} \cdots P_k$ ($k+1$ if there is no such i); we denote it by $[\pi \ll \sigma]$.

Remark 8.4.24. A sequence σ is blocking for an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ if and only if $[\sigma \gg \pi] = k$, if and only if $[\pi \ll \sigma] = 0$.

Also, given two sequences σ_l, σ_r, the sequence $\sigma_l\sigma_r$ is blocking for π if and only if $[\sigma_l \gg \pi] \geq [\pi \ll \sigma_r]$.

For the next lemma we define a partial order on portals: $P \preceq P'$ if all blocking factors of P' are also blocking factors of P. We write $\succeq$ for the reverse relation, $\simeq$ for the equivalence relation $\preceq \cap \succeq$ and $\not\simeq$ for the complement relation of $\simeq$.

Additionally, given an SCC-path $\pi = P_0 \xrightarrow{x_1} \ldots P_k$ and two sequences of positional words σ_l, σ_r, we say that the portal P_i *survives* (σ_l, σ_r) *in* π if $[\sigma_l \gg \pi] < i < [\pi \ll \sigma_r]$.

Definition 8.4.25. *Let P be a portal and σ_l and σ_r sequences of positional words.*

We define three properties that those objects may have:

P1) *$\sigma_l\sigma_r$ is not blocking for $\mathcal{A}$*
P2) *P has infinitely many minimal blocking factors*
P3) *for any accepting SCC-path π in $\mathcal{A}$, every portal in π which survives (σ_l, σ_r) is $\simeq$-equivalent to P.*

Lemma 8.4.26. *If $\mathcal{A}$ has infinitely many minimal blocking sequences, then there exist a portal P and sequences σ_l and σ_r satisfying properties P1, P2 and P3.*

Proof. By Lemma 8.4.23, a minimal blocking sequence has a bounded number of elements. Therefore, if $\mathcal{A}$ as an infinite number of minimal blocking sequences, there exists an integer i_* and an infinite family $(\sigma_j)_{j\in\mathbb{N}}$ of minimal blocking sequences of $\mathcal{A}$ such that the length of i_*-th term of σ_j is at least j, for every j. For each j, let $\sigma_{j,l}$ denote the sequence containing the elements of σ_j, up to index i_*-1, and let $\sigma_{j,r}$ denote the sequence with the elements starting from index i_*+1. As there is a finite number of SCC-paths in $\mathcal{A}$, we can extract from the sequence $(\sigma_j)_j$ an infinite subsequence $(\sigma'_j)_{j\in\mathbb{N}}$ such that for all SCC-paths π of $\mathcal{A}$, all of the $\sigma_{j,l}$ have the same left effect as $\sigma_l = \sigma_{0,l}$ on π, and symmetrically for the right effect of the $(\sigma_{j,r})_j$ and $\sigma_r = \sigma_{0,r}$.

Then, we can replace $\sigma_{j,l}$ with σ_l and $\sigma_{j,r}$ with σ_r in each σ'_j, to obtain an infinite sequence of minimal blocking sequences of the form $(\sigma_l, \nu_j, \sigma_r)$, where each ν_j is a positional word of length at least j. As these blocking sequences are minimal, the pair (σ_l, σ_r) is not blocking for $\mathcal{A}$, there is an accepting SCC-path π_* and a portal P_* that survives (σ_l, σ_r) in that π_*. If there are multiple possible choices for π_* and P_*, we choose them so that P_* is $\preceq$-minimal among the possible choices. The following claim shows that we can choose such a P_* with infinitely many minimal blocking factors.

▷ Claim 8.4.27. There exists such a P_* with infinitely many minimal blocking factors.

Claim proof. The word ν_j is blocking for all portals that survive (σ_l, σ_r), and there are arbitrarily long ν_j such that $(\sigma_l, \nu_j, \sigma_r)$ is a minimal blocking sequence. Therefore, all letters in each ν_j must belong to a minimal blocking factor of some $\preceq$-minimal portal, hence one of them has infinitely many minimal blocking factors. ◁

So far, properties P1 and P2 are satisfied. Next, we extend the sequences σ_l and σ_r until the property P3 is satisfied, while preserving properties P1 and P2.

▷ Claim 8.4.28. There exist σ_l, σ_r such that $\sigma_l\sigma_r$ is not a blocking sequence for $\mathcal{A}$, and for any accepting SCC-path π in $\mathcal{A}$, every surviving portal in π is $\simeq$-equivalent to P_*.

Claim proof. Note that for each $P \not\simeq P_*$, we can pick a positional word τ_P that is blocking for P but not for P_*, since P_* is $\preceq$-minimal.

We extend σ_l and σ_r as follows. While there is a surviving portal P that is not $\simeq$-equivalent to P_*:

- We pick an SCC-path $\pi = P_0 \xrightarrow{a_1} \dots P_k$ such that P survives in π.
- Let $i_\ell = [\sigma_l \gg \pi]$ and $i_r = [\pi \ll \sigma_r]$
- If for all $i \in \{i_\ell + 1, \dots, i_r - 1\}$, $P_i \not\simeq P_*$ then we append at the end of σ_l the sequence $\tau_{P_{i_\ell+1}}, \dots, \tau_{P_{i_r-1}}$. The sequence $\sigma_l\sigma_r$ is now blocking for π. On the other hand, since we did not add any blocking factor for P_*, there must still be a surviving portal that is $\simeq$-equivalent to it.
- If there is an $i \in \{i_\ell + 1, \dots, i_r - 1\}$ such that $P_i \simeq P_*$ then let c be the maximal index in $\{i_\ell + 1, \dots, i\}$ such that P_c is not equivalent to P_* for $\simeq$, or i_ℓ if there is no such index. Symmetrically, let d the minimal index in $\{i, \dots, i_r - 1\}$ such that $P_d \not\simeq P_*$, or i_r if there is no such index. We append at the end of σ_l the sequence $\tau_{P_{i_\ell+1}}, \dots, \tau_{P_c}$. We append at the beginning of σ_r the sequence $\tau_{P_d}, \dots, \tau_{P_{i_r-1}}$. Now all surviving portals in π are $\simeq$-equivalent to P_*, and P_i still survives.

We iterate this step until all surviving portals are $\simeq$-equivalent to P_*. We made sure that at least one portal was still surviving after each step, hence in the end the sequence $\sigma_l\sigma_r$ is not blocking for $\mathcal{A}$. ◁

□

Lemma 8.4.29. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_\ell$ be an accepting SCC-path, denote $P_j = s_j, x_j \rightsquigarrow t_j, y_j$ for each $j = 0, \dots, \ell$, let $i \in \{0, \dots, \ell\}$, and let $\sigma_l = (\nu_{1,l}, \dots, \nu_{k,l})$ be a sequence such that $[\sigma_l \gg \pi] < i$.*

Then, for any integer $N \in \mathbb{N}$, there is a positional word w_l^* of length at most $(3|\mathcal{A}|^3 + |\mathcal{A}|)(k+1) + N(2p^2+p)k|\mathcal{A}| + pN\sum_{t=1}^{k} |\nu_{t,l}|$ such that $|w_l^*| = x_i - x_0 \pmod p$, there is a run reading w_l^* from s_0 to s_i in $\mathcal{A}$, and $(x_0 : w_l^*)$ contains N occurrences of $\nu_{1,l}$, followed by N occurrences of $\nu_{2,l}$, etc. up to $\nu_{k,l}$, all disjoint.

Proof. We define w_l^* by induction on k, the length of σ_l. As π is accepting, by definition its language $\mathcal{L}(\pi)$ is nonempty, and thus for all $j \in \{0, \dots, \ell\}$, there exists a word u_j of length $y_j - x_j \pmod p$ that labels a path from s_j to t_j. By Fact 8.2.4, there is such a word u_j of length at most $3|\mathcal{A}|^2$. As a result, for all $z \in \{0, \dots, \ell\}$ we can form a word $w_z = u_0 a_1 u_1 \cdots a_z$, of length at most $3|\mathcal{A}|^3 + |\mathcal{A}|$, that labels a path of length $x_z - x_0 \pmod p$ from q_0 to s_z in $\mathcal{A}$. If $k = 0$, we can simply set $w_l^* = w_i$.

Let $k > 0$, and assume that the lemma holds for $k-1$. Let $j = [\nu_{1,l} \gg \pi]$. As $[\nu_{1,l} \gg \pi] \leq [\sigma_l \gg \pi] < i$, we have $j < i$, hence $\nu_{1,l}$ is not blocking for P_{j+1}. As a consequence, there is a word v_j that labels a path from s_j to t_j such that $\tau_j = (x_j : v_j)$ has $\nu_{1,l}$ as a factor. We can remove cycles of length $0 \pmod p$ in that path, before and after reading τ_j, so we can assume that $|v_j| \leq |\nu_{1,l}| + 2p|\mathcal{A}|$. As s_j and t_j are in the same SCC, we can extend v_j into a word v'_j of length at most $|v_j| + |\mathcal{A}| \leq |\nu_{1,l}| + (2p+1)|\mathcal{A}|$ that labels a cycle from s_j to itself.

Let $\sigma' = (\nu_{2,l}, \dots, \nu_{k,l})$ and $\pi' = P_{j+1} \xrightarrow{a_{j+2}} \cdots P_\ell$. As σ_l is the concatenation of $\nu_{1,l}$ and σ', and $j = [\nu_{1,l} \gg \pi]$, we have $[\sigma' \gg \pi'] < i - j - 1$. By induction hypothesis, there is a word w' of length at most $(3|\mathcal{A}|^3 + |\mathcal{A}|)k + N(2p^2+p)(k-1)|\mathcal{A}| + pN\sum_{t=2}^{k} |\nu_{t,l}|$ such that $|w'| = x_i - x_{j+1} \pmod p$, there is a run reading w' from s_{j+1} to s_i in $\mathcal{A}$, and $(x_{j+1} : w')$ contains N occurrences of $\nu_{t,l}$, all disjoint, for each $t = 2, \dots, k$.

We set $w_l^* = w_{j+1}(v'_j)^{pN} w'$. This word has length $x_i - x_0 \pmod p$, and satisfies:

$$\begin{aligned}
|w_l^*| &\leq |w_{j+1}| + pN|v'_j| + |w'| \\
&\leq 3|\mathcal{A}|^3 + |\mathcal{A}| + pN(|\nu_{1,l}| + (2p+1)|\mathcal{A}|) + |w'| \\
&\leq (3|\mathcal{A}|^3 + |\mathcal{A}|)(k+1) + N(2p^2+p)k|\mathcal{A}| + pN\sum_{t=1}^{k} |\nu_{t,l}|.
\end{aligned}$$

By construction, the word $(x_0 : w_l^*)$ labels a path from s_0 to s_i, and contains N occurrences of $\nu_{1,l}$, followed by N occurrence of $\nu_{2,l}$, etc. up to $\nu_{k,l}$, all disjoint, which concludes the proof. □

Lemma 8.4.30. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_\ell$ be an accepting SCC-path, denote $P_j = s_j, x_j \rightsquigarrow t_j, y_j$ for each $j = 0, \dots, \ell$, let $i \in \{0, \dots, \ell\}$, and let $\sigma_r = (\nu_{1,r}, \dots, \nu_{k,r})$ be a sequence such that $[\sigma_l \gg \pi] < i$.*

Then, for any integer $N \in \mathbb{N}$, there is a word w_r^* of length at most $(3|\mathcal{A}|^3 + |\mathcal{A}|)(k+1) + N(2p^2+p)k|\mathcal{A}| + pN\sum_{i=1}^{k} |\nu_{i,r}|$ such that $|w_r^*| = x_i - x_0 \pmod p$, there is a run reading w_r^* from s_0 to s_i in $\mathcal{A}$, and $(x_0 : w_r^*)$ contains N occurrences of $\nu_{1,r}$, followed by N occurrence of $\nu_{2,r}$, etc. up to $\nu_{k,r}$, all disjoint.

Proof. By a proof symmetric to the one of the previous lemma. □

Given a sequence σ, define $||\sigma||$ as the sum of the lengths of the terms of σ.

Lemma 8.4.31. *If there exist a portal P and σ_l, σ_r satisfying properties P1, P2 and P3 then $\mathcal{L}(\mathcal{A})$ is hard.*

Proof of Lemma 8.4.31. A direct consequence of properties P1 and P3 is that for all ν', then $\sigma_l \nu' \sigma_r$ is blocking for $\mathcal{A}$ if and only if ν' is blocking for P.

The proof goes as follows: we show that we can turn an algorithm testing $\mathcal{L}(\mathcal{A})$ with $f(\varepsilon)$ samples into an algorithm testing $\mathcal{L}(P)$ with $f(\varepsilon/X)$ samples with X a constant. We then apply Theorem 8.3.14 from the strongly connected case to obtain the lower bound.

Consider an algorithm testing $\mathcal{L}(\mathcal{A})$ with $f(\varepsilon)$ samples for some function f. We describe an algorithm for testing $\mathcal{L}(P)$. Say we are given a threshold ε and a word v of length n. First of all we can apply Lemmas 8.4.29 and 8.4.30 to compute two words w_l^* and w_r^* of length at most $E + \varepsilon n F$ for some constants E and F such that we can read w_l^* from q_0 to s and w_r^* from t to q_f and w_l^* contains occurrences of each element of σ_l at least εn times, all disjoint, with all occurrences of the i-th of σ_l appearing before element j for $i < j$, and similarly for w_r^* and σ_r. Let $w = w_l^* v w_r^*$, and assume that $|v| \geq \frac{6p^2|\mathcal{A}|^2}{\varepsilon}$ and that $d(v, \mathcal{L}(P)) < +\infty$.

- If $v \in \mathcal{L}(P)$ then clearly $w \in \mathcal{L}(\mathcal{A})$.
- If $d(v, \mathcal{L}(P)) \geq \varepsilon n$ then by Lemma 8.3.12 (in light of Lemma 8.4.7), $(x : v)$ contains at least $\frac{\varepsilon n}{6p^2|\mathcal{A}|^2}$ blocking factors for P. Then we have that w contains at least $\frac{\varepsilon n}{6p^2|\mathcal{A}|^2}$ disjoint blocking sequences for $\mathcal{A}$. As a result, $d(w, \mathcal{L}(\mathcal{A})) \geq \frac{\varepsilon n}{6p^2|\mathcal{A}|^2}$. We divide this by the length of w, which is at most $2E + 2F\varepsilon n + n$. We obtain that $d(w, \mathcal{L}(\mathcal{A})) \geq \frac{\varepsilon}{X}|w|$ for some constant X.

Let us now describe the algorithm for testing $\mathcal{L}(P)$.

- If $\mathcal{L}(P) \cap \Sigma^n = \emptyset$ then we reject.
- If $|v| < \frac{6p^2|\mathcal{A}|^2}{\varepsilon}$ then we read v entirely and check that it is in $\mathcal{L}(P)$.
- If $v \in \mathcal{L}(P)$ then we apply our algorithm for testing $\mathcal{L}(\mathcal{A})$ on $w = w_l^* v w_r^*$ with parameter $\varepsilon' = \frac{\varepsilon}{X}$.

The number of queries used on v is at most the number of queries needed on w, hence at most $f(\varepsilon/X)$ queries. We obtain a procedure to test $\mathcal{L}(P)$ using $f(\varepsilon/X)$ queries. By Theorem 8.3.14, $f(\varepsilon/X) = \Omega(\log(\varepsilon^{-1})/\varepsilon)$, hence $f(\varepsilon) = \Omega(\log(\varepsilon^{-1})/\varepsilon)$. This concludes our proof. □

Proposition 8.4.32. *If $\mathcal{A}$ has infinitely many minimal blocking sequences, then $\mathcal{L}(\mathcal{A})$ is hard.*

Proof. We combine Lemmas 8.4.26 and 8.4.31. □

8.5 Trivial and Easy languages

8.5.1 Upper bound for easy languages

We first establish that an automaton with finitely many minimal blocking sequences is easy (or trivial) to test.

Lemma 8.5.1. *Let $\mathcal{A}$ be an NFA with a finite number of minimal blocking sequences, let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path of $\mathcal{A}$, let $L = \mathcal{L}(\pi)$, and let μ be a positional word of length n such that $d(\mu, L)$ is finite. There are constants B, D such that if $n \geq 2D/\varepsilon$ and μ is ε-far from L, then μ can be partitioned into $\mu = \tau_0\tau_1\cdots\tau_k$ such that for every $i = 0, \ldots, k$, τ_i contains at least $\frac{\varepsilon n}{D}$ disjoint blocking factors for P_i, each of length at most B.*

Proof. By Corollary 8.4.19, the positional word μ contains $N \geq \varepsilon n/C$ disjoint blocking sequences $(\sigma_j)_{j=1,\ldots,N}$ for $\mathcal{A}$, for some constant C. We can extract from each σ_j a minimal blocking sequence $\sigma'_j = (\nu_{0,j}, \ldots, \nu_{s_j,j})$. By definition of blocking sequences, σ'_j is also blocking for π.

As $\mathcal{A}$ has a finite number of minimal blocking sequences, hence there is a constant B such that any $\nu_{i,j}$ has length at most B.

We build the decomposition $\mu = \tau_0\tau_1\cdots\tau_k$ with the following iterative process. For the index $i = 0$, we set τ_0 to the shortest prefix of μ that contains the leftmost $N/(k+1)$ components of the σ'_j that are blocking for P_0. Since the $(\sigma_j)_j$ are disjoint in μ, so are the $(\nu_{i,j})_{i,j}$, and this leaves us with at least $N(1-1/(k+1))$ of the σ'_j that have their component blocking for P_0, and therefore also for P_1 in the part of μ outside of τ_0. We then iterate again for $i = 1,\ldots,k+1$, with the invariant that at step i, we have $N(1-i/(k+1))$ of the σ'_j that have their component blocking for P_i outside for $\tau_0\ldots\tau_{i-1}$. We then take for τ_i the shortest prefix of the rest of μ that contains the leftmost $N/(k+1)$ components of these σ'_j that are blocking for P_i.

At each step, the factor τ_i contains $N/(k+1)$ blocking factors for P_i, hence the decomposition $\mu = \tau_0\tau_1\cdots\tau_k$ has the desired property for $D = C\cdot(k+1)$. □

Corollary 8.5.2. *If $\mathcal{A}$ has finitely many minimal blocking sequences, then there is a tester for $\mathcal{L}(\mathcal{A})$ that uses $O(1/\varepsilon)$ queries.*

Proof. We use the same algorithm that for Theorem 8.4.16, except that we use the factors given by Lemma 8.5.1, therefore, in the call to the SAMPLER function (Algorithm 3), the upper bound on the length of the factors is B instead of $O(1/\varepsilon)$. In that case, the query complexity becomes $O(\log(B)/\varepsilon) = O(1/\varepsilon)$. □

This already gives us a clear dichotomy: all languages either require $\Theta(\log(\varepsilon^{-1})/\varepsilon)$ queries to be tested, or can be tested with $O(1/\varepsilon)$ queries.

8.5.2 Separation between trivial and easy languages

It remains to show that languages that can be tested with $O(1/\varepsilon)$ queries have query complexity either $\Theta(1/\varepsilon)$, or 0 for large enough n. Our proof uses the class of *trivial* regular languages identified by Alon et al. [24], which we revisit next.

An example of a trivial language is L_2 consisting of words containing at least one a over the alphabet $\{a,b\}$. For any word u, replacing any letter by a yields a word in L_2, hence $d(u,L_2)\leq 1$. Therefore, for $n > 1/\varepsilon$, no word of length n is ε-far from L_2, and the trivial property tester that answers "yes" without sampling any letter is correct.

Alon et al. [24] define non-trivial languages as follows.

Definition 8.5.3 ([24, Definition 3.1])**.** *A language L is non-trivial if there exists a constant $\varepsilon_0 > 0$, so that for infinitely many values of n the set $L\cap\Sigma^n$ is non-empty, and there exists a word $w\in\Sigma^n$ so that $d(w,L)\geq\varepsilon_0 n$.*

It is easy to see that if a language is trivial in the above sense (i.e. not non-trivial), then for large enough input length n, the answer to testing membership in L only depends n, and the algorithm does not need to query the input. Alon et al. [24, Property 2] show that if a language is non-trivial, then testing it requires $\Omega(1/\varepsilon)$ queries for small enough $\varepsilon > 0$.

To obtain our characterization of *trivial* languages, we show that $\mathsf{MBS}(\mathcal{A})$ is non-empty if and only if $\mathcal{L}(\mathcal{A})$ is non-trivial (in the above sense). It follows that if $\mathsf{MBS}(\mathcal{A})$ is empty, then testing $\mathcal{L}(\mathcal{A})$ requires 0 queries for large enough n. Furthermore, by the result of Alon et al. [24], if $\mathsf{MBS}(\mathcal{A})$ is non-empty, then testing $\mathcal{L}(\mathcal{A})$ requires $\Omega(1/\varepsilon)$ queries.

Recall that we focus on infinite languages, since we know that all finite ones are trivial (Remark 8.1.4).

Lemma 8.5.4. *$\mathsf{MBS}(\mathcal{A})$ is empty if and only if $L = \mathcal{L}(\mathcal{A})$ is trivial.*

We prove the two directions separately.

Lemma 8.5.5. *If $\mathsf{MBS}(\mathcal{A})$ is empty, then $L = \mathcal{L}(\mathcal{A})$ is trivial in the sense of Definition 8.5.3.*

Proof. We showed in Corollary 8.4.19 that if μ is long enough and ε-far from L, then μ contains $\Omega(\varepsilon n)$ disjoint blocking sequences for $\mathcal{A}$. As $\mathcal{A}$ has no minimal blocking sequences, it does not have blocking sequences either, and long enough words cannot be ε-far from L, hence it is trivial in the sense of Definition 8.5.3. □

To prove the converse property, we need the following extension of Kleene's Lemma for languages of SCC-paths: for large enough ℓ, whether $\mathcal{L}(\pi)$ contains a word of length ℓ only depends on the value of ℓ modulo p (p is the lcm of all the lengths of the simple cycles in $\mathcal{A}$).

Lemma 8.5.6. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ be an SCC-path. There exists a constant B such that, for all $\ell \geq B$, if there is a word μ of length ℓ in $\mathcal{L}(\pi)$, then there exists a word μ' of length $\ell - p$ and a word μ'' of length $\ell + p$ in $\mathcal{L}(\pi)$.*

Proof. Recall the definition of $\mathcal{L}(\pi)$ (Definition 8.4.9):

$$\mathcal{L}(\pi) = L_0 a_1 L_1 a_2 \cdots L_k, \text{ where } L_i = \mathcal{L}(P_i) \text{ for } i = 0, \ldots, k.$$

It follows that a word $\mu \in \mathcal{L}(\pi)$ can be written as $\mu = \mu_1 a_1 \mu_2 \ldots \mu_k$ with $\mu_i \in L_i$. Each L_i is recognized by a strongly connected automaton $\mathcal{A}_i$ with at most $p|\mathcal{A}|$ states. Let $B = 5(p|\mathcal{A}|)^2$. If the length ℓ of μ exceeds B, then the run of μ in each of the $\mathcal{A}_i$'s contains simple loops with sum of lengths greater than $p + 3(p|\mathcal{A}|)^2$. Let $\ell_0 + p$ denote the sum of the length of these simple cycles: by construction ℓ_0 is greater than $3(p|\mathcal{A}|)^2$. We remove these simple cycles from the run: the resulting run is still in $\mathcal{L}(\pi)$. Next, select any non-trivial SCC S_i in π and let s be a state of S_i used by the run. As $\ell_0 \geq 3(p|\mathcal{A}|)^2$, by Fact 8.2.4, there is a path of length ℓ_0 from s to itself in $\mathcal{A}_i$. Adding this path to the run yields an accepting run of length $\ell - (\ell_0 + p) + \ell_0 = \ell - p$: the word labeling this run is the desired word μ'.

To obtain μ'', consider any simple cycle in the run of μ in $\mathcal{A}$, and let m denote the length of this cycle. By definition of p, m divides p. Iterating this cycle p/m times yields a word μ'' of length $\ell + p$ that is in $\mathcal{L}(\pi)$. □

Corollary 8.5.7. *Let π be an SCC path. For large enough ℓ, whether there is an word of length ℓ in $\mathcal{L}(\pi)$ only depends on the value of $\ell \pmod p$.*

To finish our characterization of trivial languages, we show that if $\mathsf{MBS}(\mathcal{A})$ is not empty, then $L = \mathcal{L}(\mathcal{A})$ is non-trivial in the sense of Alon et al. [24].

Lemma 8.5.8. *Let $\mathcal{A}$ be a trim NFA such that $L = \mathcal{L}(\mathcal{A})$ is infinite. If $\mathcal{A}$ admits a blocking sequence, then there exists $\varepsilon_0 > 0$, such that for infinitely many n there exist words in $\mathcal{L}(\mathcal{A}) \cap \Sigma^n$ and there exists $w \in \Sigma^n$ such that $d(w, \mathcal{L}(\mathcal{A})) \geq \varepsilon_0 n$*

Proof. Let $\sigma = (\mu_1, \ldots, \mu_k)$ be a blocking sequence for $\mathcal{A}$. We can assume w.l.o.g. that σ is strongly blocking for every accepting π of $\mathcal{A}$, as we can make it strongly blocking by concatenating σ to itself K times, where K is the maximum length of an accepting SCC-path in $\mathcal{A}$. Let C be the maximum length of a μ_i's. As L is infinite, there exists an

accepting SCC-path π in $\mathcal{A}$ and $w \in \mathcal{L}(\pi)$ with $|w| \geq t$ for arbitrary t. By Corollary 8.5.7, for all sufficiently large ℓ such that $\ell = |w| \pmod p$, there exists $w' \in \mathcal{L}(\pi)$ with $|w'| = \ell$.

For all $i = 1, \ldots, k$, let ν_i be a shortest word of the form $(0 : v_i)$, for some v_i, and of length ℓ_i equal to 0 modulo p, such that μ_i is a factor of ν_i. By minimality, ℓ_i is at most $C + 2p$. Then, for any integer $N \in \mathbb{N}$, let $w_N = \nu_1^N \cdots \nu_k^N (0 : a^{|w|})$, where a is an arbitrary letter.

As w_N is of length $|w| \pmod p$, there is a word of the same length in $\mathcal{L}(\mathcal{A})$, i.e. $\mathcal{L}(\mathcal{A}) \cap \Sigma^n$ is nonempty. On the other hand, it contains N disjoint occurrences of σ, which is a strongly blocking sequence for every accepting SCC-path of $\mathcal{A}$, therefore, the distance between w_N and $\mathcal{L}(\mathcal{A})$ is at least N. Furthermore, the length of w_N is less than $|w| + N(C + 2p)$. Therefore, if we let $\varepsilon_0 = \frac{1}{C+2p+|w|}$, then we have $\varepsilon_0 |w_N| \leq N \leq d(w_N, \mathcal{L}(\mathcal{A}))$, i.e. w_N is ε-far from L for any $\varepsilon \leq \varepsilon_0$ and any N. □

It is easy to see that if a language is trivial in the above sense, then for large enough input length n, membership in L only depends n, and the algorithm does not need to query the input. Alon et al. [24] show that if a language is non-trivial, then testing it requires $\Omega(1/\varepsilon)$ queries for small enough $\varepsilon > 0$. As a corollary of that lower bound, we obtain that if $\mathsf{MBS}(\mathcal{A})$ is non-empty, then testing $\mathcal{L}(\mathcal{A})$ requires $\Omega(1/\varepsilon)$ queries.

8.6 The Complexity of Classifying Regular Languages

In the previous sections, we have shown that testing some regular languages (*easy* ones) that requires fewer queries than testing others (*hard* ones). Therefore, given the task of testing a word for membership in $\mathcal{L}(\mathcal{A})$, it is natural to first try to determine if the language of $\mathcal{A}$ is easy, and if this is the case, run the appropriate ε-tester, that uses fewer queries. In this section, we investigate the computational complexity of checking which class of the trichotomy the language of a given automaton belongs to. We formalize this question as the following decision problems:

Problem 8.6.1 (Triviality problem)**.** Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ trivial?

Problem 8.6.2 (Easiness problem)**.** Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ easy?

Problem 8.6.3 (Hardness problem)**.** Given an finite automaton $\mathcal{A}$, is $\mathcal{L}(\mathcal{A})$ hard?

In these problems, the automaton $\mathcal{A}$ is the input and is no longer fixed. We show that, our combinatorial characterization based on minimal blocking sequences is effective, in the sense that all three problems are decidable. However, it does not lead to efficient algorithms, as both problems are PSPACE-complete.

Theorem 8.6.4. *The triviality and easiness problems are both* PSPACE*-complete, even for strongly connected NFAs.*

In Section 8.6.1 we show the PSPACE upper bounds on the hardness and triviality problems (Propositions 8.6.11 and 8.6.13). The upper bound on the easiness problem follows immediately, as the three properties form a trichotomy.

In Section 8.6.2, we show that all three problems are PSPACE-hard (Lemma 8.6.15 and Corollary 8.6.17).

8.6.1 A PSPACE upper-bound

8.6.1.1 Testing hardness

A naive algorithm to check hardness of a language $\mathcal{L}(\mathcal{A})$ would be to construct an automaton recognising blocking sequences of $\mathcal{L}(\mathcal{A})$ (exponential in $\mathcal{A}$), and use it to get an automaton recognising the minimal ones (which requires complementation and could yield another exponential blow-up). This would a priori not give a PSPACE algorithm, since we obtain a doubly-exponential state space. We solve this by providing another characterisation of automata with hard languages, resulting in a recursive PSPACE algorithm to test it.

Lemma 8.6.5. *Let $\pi = P_0 \xrightarrow{a_1} \cdots P_\ell$ be an SCC-path, i an index, Π a set of SCC-paths and $(\sigma_{\pi'})_{\pi' \in \Pi}$ a family of sequences of positional words such that $[\sigma_{\pi'} \gg \pi] < i$ for all π'.*

There exists a sequence of positional words σ such that:

- $[\sigma \gg \pi] < i$
- $[\sigma_{\pi'} \gg \pi'] \leq [\sigma \gg \pi']$ *for all* $\pi' \in \Pi$.

Proof. We prove this by induction on the sum of the lengths of the elements of Π. If Π is empty or contains only empty sequences, then we can set σ as the empty sequence.

If not, let π^* be such that the first term ν_1 of $\sigma_{\pi^*} = (\nu_1, \ldots, \nu_k)$ has the least left effect on π among all SCC-paths in Π; let $\pi^* = P'_0 \xrightarrow{a'_1} \cdots P'_{\ell'}$. We consider the effect of ν_1 (as a single-element sequence) on π^* and π: let $j = [\nu_1 \gg \pi^*]$ and $r = [\nu_1 \gg \pi]$.

Next, we build a set Π' of SCC-paths as follows. Let $\overline{\pi}$ denote the part of π^* that survives ν_1, if any, i.e. $\overline{\pi} = P'_{j+1} \xrightarrow{a'_{j+1}} \cdots P'_{\ell'}$. We define $\Pi' = \Pi \setminus \{\pi^*\} \cup \{\overline{\pi}\}$ if $j < \ell'$ and $\Pi' = \Pi \setminus \{\pi^*\}$ otherwise. In the first case the sequence associated with $\overline{\pi}$ is $\sigma_{\overline{\pi}} = (\nu_2, \ldots, \nu_k)$.

We now wish to apply the induction hypothesis to the set Π' and the part of π that survives ν_1, i.e. on $\tilde{\pi} = P_{r+1} \xrightarrow{a_{r+1}} \ldots \to P_\ell$, with a target left effect of $i - r - 1$. By construction, the sum of the lengths of the elements in Π' is smaller than that of Π. The following claim shows that, for any π' in Π', the left effect of $\sigma_{\pi'}$ on $\tilde{\pi}$ is at most $i - r - 1$.

▷ Claim 8.6.6. For all $\pi' \in \Pi'$, we have $[\sigma_{\pi'} \gg \tilde{\pi}] < i - r - 1$.

Claim proof. Let $\pi' \in \Pi \setminus \{\pi^*\}$, and let $\sigma_{\pi'} = (\nu'_1, \ldots, \nu'_m)$. Since the first term of σ_{π^*} was the one with the least left effect on π, the first term of every other sequence has a left effect at least r on it. Formally, let $z = [\nu'_1 \gg \pi]$: we have $z \geq r$.

In other words, ν'_1 is blocking for all portals in $\tilde{\pi}$ up to P_z. Therefore, the sequence $(\nu'_2, \ldots, \nu'_m)$ will be applied to the same portals in π and in $\tilde{\pi}$. Since portal P_i survives in π, it must also survive in $\tilde{\pi}$, and we have $[\sigma_{\pi'} \gg \tilde{\pi}] < i - r - 1$. ◁

By induction hypothesis, we obtain a sequence $\tilde{\sigma}$ such that

- $[\tilde{\sigma} \gg \tilde{\pi}] < i - r - 1$
- $[\sigma_{\pi'} \gg \pi'] \leq [\tilde{\sigma} \gg \pi']$ for all $\pi' \in \Pi'$.

Then, the sequence obtained by prepending ν_1 to $\tilde{\sigma}$ satisfies both conditions of the lemma, as $\tilde{\pi}$ is the part of π that survives ν_1, and prepending ν_1 cannot decrease the left effect of a sequence. □

Lemma 8.6.7. *An automaton $\mathcal{A}$ is hard if and only if there exists an accepting SCC-path π containing a portal P such that:*

- *P has infinitely many minimal blocking factors.*

- *For any accepting SCC-path π' there exist sequences $\sigma_{l,\pi'}, \sigma_{r,\pi'}$ such that:*
  - *P survives $(\sigma_{l,\pi'}, \sigma_{r,\pi'})$ in π*
  - *All portals surviving $(\sigma_{l,\pi'}, \sigma_{r,\pi'})$ in π' are $\simeq$-equivalent to P*

Proof. The left-to-right direction follows from Corollary 8.5.2, by taking $\sigma_{l,\pi'} = \sigma_l$ and $\sigma_{r,\pi'} = \sigma_r$ for every π'.

Let us now prove the other direction. Suppose we have π and P satisfying the conditions of the lemma. We only need to construct two sequences σ_l, σ_r such that properties P1 and P3 are satisfied. The result follows by Lemma 8.4.31.

Let Π be the set of accepting SCC-paths in $\mathcal{A}$. Consider families of sequences $(\sigma_{l,\pi'})_{\pi' \in \Pi}$ and $(\sigma_{r,\pi'})_{\pi' \in \Pi}$ such that for all $\pi' \in \Pi$:

- P survives $(\sigma_{l,\pi'}, \sigma_{r,\pi'})$ in π
- All portals surviving $(\sigma_{l,\pi'}, \sigma_{r,\pi'})$ in π' are $\simeq$-equivalent to P

Let i be the index of P in π. By Lemma 8.6.5 we can build a sequence σ_l such that

- $[\sigma_l \gg \pi] < i$, and
- $[\sigma_{l,\pi'} \gg \pi'] \leq [\sigma_l \gg \pi']$ for all $\pi' \in \Pi$.

Using a symmetric argument, we build a sequence σ_r such that

- $i < [\pi \ll \sigma_r]$, and
- $[\pi' \ll \sigma_{r,\pi'}] \geq [\pi' \ll \sigma_r]$ for all $\pi' \in \Pi$.

As a consequence, for all accepting SCC-path $\pi' \in \Pi$, all portals surviving (σ_l, σ_r) in π' are $\simeq$-equivalent to P. Furthermore, P survives (σ_l, σ_r) in π.

We have shown that P and (σ_l, σ_r) satisfy properties P1 and P3. P2 is immediate by assumption. We simply apply Lemma 8.4.31 to obtain the result. □

Next, we establish that the items listed in the previous lemma can all be checked in polynomial space in $|\mathcal{A}|$.

Lemma 8.6.8. *Given a portal P, we can check whether it has infinitely many minimal blocking factors in space polynomial in $|\mathcal{A}|$.*

Proof. Recall that, by Lemma 8.4.7, $L = \mathcal{L}(P)$ is recognized by a strongly connected automaton $\mathcal{A}'$ with at most $p|\mathcal{A}|$ states. While this number may be exponential in $|\mathcal{A}|$, the transition function of $\mathcal{A}'$ can be computed in polynomial space from the polynomial-sized representation of a state. Furthermore, in this case, we can show that the same property holds for the construction used in Lemma 8.3.15, as in the determinization step, all states share the index modulo p.

We then simply need to check if the resulting automaton has an infinite language, which is the case if and only if it has a cycle reachable from the initial state and from which a final state is reachable. This can be checked by exploring the state space of the automaton, in non-deterministic polynomial space (in $|\mathcal{A}|$), and applying Savitch's theorem [268, Theorem 1], which states that PSPACE = NPSPACE. □

Lemma 8.6.9. *Given two SCC-paths π and π', one can check in* PSPACE *whether there is a sequence σ that is blocking for π and not π'.*

Proof. The algorithm relies on the following property.

▷ Claim 8.6.10. There is a sequence σ that is blocking for $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ and not $\pi' = P'_0 \xrightarrow{a'_1} \cdots P'_\ell$ if and only if either:

- there is a positional word μ that is a blocking factor for P_0 and not P'_0 and there is a sequence σ' that is blocking for $P_1 \xrightarrow{a_2} \cdots P_k$ and not π',

- or there is a positional word μ that is a blocking factor for P_0 and P'_0 and there is a sequence σ' that is blocking for $P_1 \xrightarrow{a_2} \cdots P_k$ and not $P'_1 \xrightarrow{a'_2} \cdots P'_\ell$.

Claim proof. The right-to-left direction is clear (just take $\sigma = \mu\sigma'$ in both cases).

For the left-to-right direction, consider a sequence σ that is blocking for π and not π', of minimal length. Let σ_+ and μ be such that $\sigma = \mu\sigma_+$.

- If μ is not blocking for P_0 then σ_+ is blocking for π and not π', contradicting the minimality of σ.
- If μ is blocking for P_0 and not P'_0 then we set $\sigma' = \sigma$. We know that σ is not blocking for π'. On the other hand, as σ is blocking for π, it is also blocking for $P_1 \xrightarrow{a_2} \cdots P_k$.
- If μ is blocking for both P_0 and P'_0 then we set $\sigma' = \sigma$. As σ is blocking for π, it is also blocking for $P_1 \xrightarrow{a_2} \cdots P_k$. On the other hand, if σ was blocking for $P'_1 \xrightarrow{a'_2} \cdots P'_\ell$, then it would also be blocking for π', a contradiction. Hence σ is not blocking for $P'_1 \xrightarrow{a'_2} \cdots P'_\ell$

◁

The claim above lets us define a recursive algorithm.

- First check if there is a positional word μ that is blocking for P_0 and not P'_0. If it is the case, make a recursive call to check if there is a sequence σ' that is blocking for $P_1 \xrightarrow{a_2} \cdots P_k$ and not π'. If it is the case, answer yes.
- Then check if there is a positional word μ that is a blocking factor for P_0 and P'_0. If so, make a recursive call to check if there is a sequence σ' that is blocking for $P_1 \xrightarrow{a_2} \cdots P_k$ and not $P'_1 \xrightarrow{a'_2} \cdots P'_\ell$. If it is the case, answer yes.

If both items fail, answer no.

The existence of those positional words can be checked in polynomial space using the automaton $\mathcal{B}$ constructed in the proof of Lemma 8.6.8. The depth of the recursive calls is at most the sum of the lengths of π and π', which is bounded by $2|\mathcal{A}|$. In consequence, this algorithm runs in polynomial space.

□

Proposition 8.6.11. *The hardness problem is in* PSPACE.

Proof. Our algorithm is based on Lemma 8.6.7. We use the following algorithm to check whether the characterization holds.

1. First, we nondeterministically guess an SCC-path $\pi = P_0 \xrightarrow{a_1} \cdots P_k$ and an index i.
2. Using Lemma 8.6.8, we check that P_i has infinitely many minimal blocking factors.
3. For each accepting SCC-path $\pi' = P'_0 \xrightarrow{a'_1} \cdots P'_\ell$ of $\mathcal{A}$, we guess indices j_l and j_r, and check that every portal P'_j with $j_l < j < j_r$ is $\simeq$-equivalent to P_i.
4. Then, we use Lemma 8.6.9 to check that there is a sequence σ_l that is blocking for $P'_0 \xrightarrow{a'_1} \cdots P'_{j_l}$ and not $P_0 \xrightarrow{a_1} \cdots P_i$. Symmetrically, we check that there is a sequence σ_r that is blocking for $P'_{j_r} \xrightarrow{a'_1} \cdots P'_\ell$ and not $P_i \xrightarrow{a_{i+1}} \cdots P_k$.

If all those tests succeed, we answer "yes", otherwise we answer "no". This algorithm is correct and complete by Lemma 8.6.7. □

8.6.1.2 Testing triviality

We show the PSPACE upper bound on the complexity of checking if a language is trivial. It is based on the characterisation of trivial languages given by Lemma 8.5.8, and uses the following result.

Lemma 8.6.12. *Given a portal P, we can check whether it has a blocking factor in space polynomial in $|\mathcal{A}|$.*

Proof. We proceed as in the proof of Lemma 8.6.8, except that we only need to check whether some final state is reachable from the final state. □

Proposition 8.6.13. *The triviality problem is in* PSPACE.

Proof. Recall that $\mathcal{L}(\mathcal{A})$ is trivial if and only if $\mathcal{A}$ has no blocking sequences.

▷ Claim 8.6.14. There is an accepting SCC-path π of $\mathcal{A}$ that contains a portal P with no blocking factors if and only if $\mathcal{A}$ has no blocking sequence.

Claim proof. Any blocking sequence of $\mathcal{A}$ is blocking for π, therefore it contains a blocking factor for P. ◁

Therefore, it suffices to enumerate all accepting SCC-paths π in the automaton, and then check that all portals in π have at least one blocking factor, using Lemma 8.6.12. □

8.6.2 Hardness of classifying automata

We prove hardness of the triviality problem and easiness problems, concluding on their PSPACE-completeness. We reduce from the universality problem for NFAs, which is well-known to be PSPACE-complete (see e.g. [17, Theorem 10.14]).

Lemma 8.6.15. *The triviality and hardness problems are* PSPACE*-hard.*

Proof. Consider an NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ on an alphabet Σ. Without loss of generality, we assume that $\mathcal{A}$ is trim (up to removing unreachable or non-co-reachable states) and that it accepts all words of length less than 2: this can be checked in polynomial time and does not affect the PSPACE-hardness of universality. Let # and ! be two letters that are not in Σ. We apply the following transformations to $\mathcal{A}$:

- add a transition labeled by ! from every final state to the initial state q_0
- add a self-loop labeled by # to each state.

We call the resulting automaton $\mathcal{B} = (Q, \Sigma \cup \{!, \#\}, \delta', q_0, F)$. Note that $\mathcal{B}$ is strongly connected: consider any two states $q, q' \in Q$, we show that q' is reachable from q. As $\mathcal{A}$ is trim, there exists $q_f \in F$ that is reachable from q, and q' is reachable from the initial state q_0. Furthermore, we have put a ! transition from q_f to q_0, hence q' is reachable from q.

Recall that the language of a strongly connected automaton is trivial if and only if it has no minimal blocking factor, and hard if and only if it has infinitely many minimal blocking factors.

Hence, to complete the proof, we show that $\mathsf{MBF}(\mathcal{B})$ is empty when $\mathcal{A}$ is universal and infinite otherwise.

First, let us describe the language recognized by $\mathcal{B}$. It is given by

$$\mathcal{L}(\mathcal{B}) = \{u_1!u_2!\cdots!u_n \mid \forall i, u_i \in (\Sigma \cup \{\#\})^* \ \wedge \ \pi_\Sigma(u_i) \in \mathcal{L}(\mathcal{A})\},$$

where $\pi_\Sigma(u)$ is the word in Σ^* obtained by removing all letters not in Σ from u.

▷ Claim 8.6.16. If $\mathcal{A}$ is universal, then $\mathcal{B}$ is also universal.

Claim proof. Indeed, any word in u in can be uniquely decomposed into $u = u_1!u_2!\cdots!u_n$ where each u_i does not contain the letter "!". As # is idempotent on $\mathcal{B}$, $\delta'(q_0, u_i)$ is equal to $\delta(q_0, \pi_\Sigma(u_i))$ for every i. Since $\mathcal{A}$ is universal, each of the $\delta'(q_0, u_i)$ contains a final state, hence $\delta'(q_0, u_i!) = \{q_0\}$. Therefore, the set $\delta'(q_0, u)$ is equal to $\delta'(q_0, u_n)$, which contains a final state, and u is in $\mathcal{L}(\mathcal{B})$, which shows that $\mathcal{B}$ is universal. ◁

This shows that if $\mathcal{A}$ is universal, then $\mathsf{MBF}(\mathcal{B})$ is empty.

Now we show that a word $w \in \Sigma^*$ not in $\mathcal{L}(\mathcal{A})$ induces infinitely many minimal blocking factors for $\mathcal{B}$. Consider such a w of minimal size. As we assumed that $\mathcal{A}$ accepts all words of size less than 2, $|w| \geq 2$. Let u, v be words of length at least 1 such that $w = uv$. For all $n \in \mathbb{N}$, at least one of $u\#^n v, !u\#^n v, u\#^n v!, !u\#^n v!$ is a minimal blocking factor (depending respectively on whether w is not a factor of any word of $\mathcal{L}(\mathcal{A})$ or is a prefix/suffix of a word of $\mathcal{L}(\mathcal{A})$ or not). As a consequence, $\mathcal{B}$ has infinitely many blocking factors, and is thus hard to test by Theorem 8.3.2.

In summary, $\mathcal{A}$ is universal if and only if $\mathcal{B}$ is trivial to test, and $\mathcal{A}$ is *not* universal if and only if $\mathcal{B}$ is hard to test. This shows the PSPACE-hardness of the triviality problem. □

The above proof can be extended to show the PSPACE-hardness of the easiness problem.

Corollary 8.6.17. *The easiness problem is* PSPACE*-hard.*

Proof. We proceed as in the proof of Lemma 8.6.15: given an automaton $\mathcal{A}$ over an alphabet Σ, we build an automaton $\mathcal{B}$ over the alphabet $\Sigma \cup \{!, \#\}$ such that if $\mathcal{A}$ is universal, $\mathsf{MBF}(\mathcal{B})$ is empty, and if $\mathcal{A}$ is not universal, then $\mathsf{MBF}(\mathcal{B})$ is infinite.

To show the hardness of the easiness problem, let $\flat$ denote a new letter not in $\Sigma \cup \{\#, !\}$ and consider the automaton $\mathcal{B}'$ equal to $\mathcal{B}$ but taken over the alphabet $\Sigma \cup \{\#, !, \flat\}$. As there are no transitions labeled by $\flat$ in $\mathcal{B}'$, the word $\flat$ is always a minimum blocking factor of $\mathcal{B}'$. As a result, we have $\mathsf{MBF}(\mathcal{B}') = \mathsf{MBF}(\mathcal{B}) \cup \{\flat\}$, hence $\mathcal{A}$ is universal if and only if $\mathsf{MBF}(\mathcal{B}')$ is finite but non-empty: by Theorem 8.3.2, this is equivalent to $\mathcal{L}(\mathcal{B}')$ is easy to test. Therefore, the easiness problem is also PSPACE-hard. □

This concludes the proof of Theorem 8.6.4

Chapter 9

Online Language Distance to Palindromes and Squares

9.1 Introduction

The *language distance problem* is one of the most fundamental problems in formal language theory. In this problem, the task is to compute the minimal distance between a given input string S and any string of a formal language L. Introduced in the early 1970s by Aho and Peterson [18], the language distance problem has been studied extensively for regular languages under Hamming and edit distances [57], for general context-free languages, focusing mainly on the edit distance [11, 18, 79, 98, 228, 242, 261, 264, 265, 267], and the Dyck language (the language of well-nested parentheses sequences) [11, 43, 79, 98, 120, 126, 144, 218, 219, 263, 264, 265].

9.1.1 Our results.

In this chapter, we study the complexity of the *online* and *low-distance* version of the language distance problem. In the online version, where we are given a string T of length n, and the task is to compute the minimum distance from *every* prefix of T to a formal language L (the distance and the language are specified in the problem definition). In the low-distance regime, we are given a threshold parameter k, and either report that the distance is larger than k, or report the distance if it does not exceed k. In this chapter, we consider the language distance problem for both the edit distance and the Hamming distance (see Section 9.2 for definitions). We study the problem for two classical languages: the language PAL of all palindromes, where a palindrome is a string that is equal to its reversed copy, and the language SQ of all squares, where a square is the concatenation of two copies of a string. These two languages are very similar yet very different in nature: PAL is not regular but is context-free, whereas SQ is not even context-free. Formally, the problems we consider are defined as follows:

Problem 9.1.1 (k-LHD-PAL (resp. k-LHD-SQ))**.**
▷ **Input:** A string T of length n and a positive integer k.
▷ **Output:** For each $1 \leq i \leq n$, report $\min\{k+1, hd_i\}$, where hd_i is the minimum Hamming distance between $T[1..i]$ and a string in PAL (resp. in SQ).

Problem 9.1.2 (k-LED-PAL (resp. k-LED-SQ))**.**
▷ **Input:** A string T of length n and a positive integer k.

▷ **Output:** For each $1 \leq i \leq n$, report $\min\{k+1, ed_i\}$, where ed_i is the minimum edit distance between $T[1..i]$ and a string in PAL (resp. in SQ).

Amir and Porat [31] showed that there is a randomized streaming algorithm that solves the k-LHD-PAL problem in $\tilde{O}(k)$ space and $\tilde{O}(k^2)$ time per input character. We continue their line of research and give streaming algorithms using $\text{poly}(k, \log n)$ time per character and $\text{poly}(k, \log n)$ space for all four problems. As a corollary, we obtain new streaming algorithms for approximating the maximal length of a substring of a given text that is close to PAL in a text. The problem is formally defined as follows:

Problem 9.1.3 ($(1+\varepsilon)$-k-HD-PAL (resp. $(1+\varepsilon)$-k-ED-PAL))**.**
▷ **Input:** A string T of length n, a positive integer k and a real number $\varepsilon > 0$.
▷ **Output:** An integer ℓ such that $\ell^* \geq \ell \geq \ell^*/(1+\varepsilon)$, where ℓ^* is the maximum length of a substring of T that is within Hamming (resp. edit) distance k of PAL.

The previous best algorithm for $(1+\varepsilon)$-k-HD-PAL is by Grigorescu et al. [167]. They extended the works [60, 161] that studied the question of computing the length of a maximal substring of a stream that belongs to PAL and gave an algorithm that solves $(1+\varepsilon)$-k-HD-PAL using $O(\frac{k \log^9 n}{\varepsilon \log(1+\varepsilon)})$ time per character and $O(\frac{k \log^7 n}{\varepsilon \log(1+\varepsilon)})$ space. We give an algorithm that uses $O((k/\varepsilon)\log^4 n)$ time per character and $O((k/\varepsilon)\log^2 n)$ bits of space, a significant improvement over the result of Grigorescu et al. [167]. We also give the first streaming algorithm for $(1+\varepsilon)$-k-ED-PAL which uses $\tilde{O}(k^2/\varepsilon)$ time per character and bits of space.

While streaming algorithms are extremely efficient (in particular, the above space complexities account for *all* the space used by the algorithms, including the space needed to store information about the input), they are randomized by nature, which means that there is a small probability that they may produce incorrect results. Motivated by this, we also study the problems in the read-only model, where random access to the input is allowed (and not accounted for in the space usage, e.g. we are given a pointer to the input, and are not allowed to modify it). In this model, we show *deterministic* algorithms for the four problems that use $\text{poly}(k, \log n)$ time per character and $\text{poly}(k, \log n)$ *extra* space (not accounting for the input); see Table 9.1 for a summary. As a side result of independent interest, we develop the first *deterministic* read-only algorithms for computing k-mismatch and k-edit occurrences of a pattern in a text using $\text{poly}(k, \log n)$ space.

9.1.2 Related work

Offline model. In the classical *offline* model, the problem of finding *all maximal substrings* that are within Hamming distance k from PAL can be solved in $O(nk)$ time as a simple application of the kangaroo jumps technique [147]. For the edit distance, Porto and Barbosa [251] showed an $O(nk^2)$ solution. For the SQ language, the best known solutions take $O(nk \log k + \text{output})$ time for the Hamming distance [210] and $O(nk \log^2 k + \text{output})$ for the edit distance [223, 271, 272].

Online model. The k-LHD-PAL and k-LED-PAL problems can be viewed as a generalization of the classical online palindrome recognition problem (see [146] and references therein).

Problem	Model	Time (per char.)	Space	Reference
k-LHD-PAL	Streaming	$O(k \log^3 n)$	$O(k \log n)$	Thm 9.3.2
k-LHD-SQ	Streaming	$\tilde{O}(k)$	$O(k \log^2 n)$	Thm 9.3.15
k-LHD-PAL	Read-only	$O(k \log n)$	$O(k \log n)$	Thm 9.4.8
k-LHD-SQ	Read-only	$O(k \log n)$	$O(k \log n)$	Thm 9.4.10
k-LED-PAL	Streaming	$\tilde{O}(k^2)$	$\tilde{O}(k^2)$	Thm 9.3.9
k-LED-SQ	Streaming	$\tilde{O}(k^2)$	$\tilde{O}(k^2)$	Thm 9.3.23
k-LED-PAL	Read-only	$\tilde{O}(k^4)$ (amort.)	$\tilde{O}(k^4)$	Thm 9.5.9
k-LED-SQ	Read-only	$\tilde{O}(k^4)$ (amort.)	$\tilde{O}(k^4)$	Thm 9.5.11
$(1+\varepsilon)$-k-HD-PAL	Streaming	$O((k/\varepsilon) \log^4 n)$	$O((k/\varepsilon) \log^2 n)$	Cor 9.3.12
$(1+\varepsilon)$-k-ED-PAL	Streaming	$\tilde{O}(k^2/\varepsilon)$	$\tilde{O}(k^2/\varepsilon)$	Cor 9.3.14

Table 9.1: Summary of the complexities of the algorithms introduced in this chapter.

9.1.3 Technical overview

9.1.3.1 Hamming distance problems.

Our first step is to show that the Hamming distance from a string U to PAL or SQ can be expressed in terms of the distance between substrings of U or its reversal. Using the small-space sketches of Clifford et al. [107] to compute these distances, we obtain small-space streaming algorithms for k-LHD-PAL and $(1+\varepsilon)$-k-HD-PAL.

Furthermore, using small-space pattern-matching algorithms together with the structural properties of the k-mismatch occurrences of a pattern in a text [89], we develop space-efficient filtering-based algorithms for k-LHD-SQ (streaming and read-only) and k-LHD-PAL (read-only).

9.1.3.2 Edit distance problems.

Informally, the *edit distance* between two strings U and V, denoted by $\mathsf{ed}(U, V)$, is the minimum number of character insertions, deletions, and substitutions required to transform U into V. Similar to the Hamming distance, we show that the edit distance from a string U to PAL or SQ can be expressed in terms of "self-similarity" of U. This allows us to use similar approaches as for the Hamming distance problems, where tools for the Hamming distance are replaced by appropriate tools for the edit distance.

First, by replacing the Hamming distance sketch with the edit distance sketch of Bhattacharya and Koucký [61], we obtain streaming algorithms for k-LED-PAL and $(1+\varepsilon)$-k-ED-PAL.

Furthermore, the results of Bhattacharya and Koucký [61] show a reduction from the edit distance to the Hamming distance via locally consistent string decompositions, which allows us to solve k-LED-SQ in streaming by reducing to k-LHD-SQ.

Finally, by replacing the online read-only algorithm for finding the k-mismatch occurrences of a pattern in a text with an online read-only algorithm for finding k-error occurrences, and replacing the structural results for the Hamming distance with the structural results for the edit distance, we obtain read-only algorithms for k-LED-PAL and

k-LED-SQ.

9.2 Preliminaries

Given two non-empty strings U, Q and an operator F defined over pairs of strings (such as a distance), we use the notation $F(U, Q^\infty)$ for the application of F to U and the prefix of $Q^\infty = QQ\cdots$ that has the same length as U, i.e., $F(U, Q^\infty) = F(U, Q^m[..|U|])$, where m is any integer such that $|Q^m| \geq |U|$. We define $F(Q^\infty, U)$ symmetrically.

9.2.1 Hamming distance, palindromes, and squares

The Hamming distance between two strings S, T (denoted $\mathsf{hd}(S, T)$) is defined to be equal to infinity if S and T have different lengths, and otherwise to the number of positions where the two strings differ (mismatches). We define the *mismatch information* between two length-n strings S and T, $\mathsf{MI}(S, T)$ as the set $\{(i, S[i], T[i]) : i \in [1..n] \text{ and } S[i] \neq T[i]\}$. For two strings P, T, a position $i \in [|P|..|T|]$ of T is a *k-mismatch occurrence* of P in T if $\mathsf{hd}(T(i - |P|..i], P) \leq k$. For an integer k, we denote $\mathsf{hd}_{\leq k}(X, Y) = \mathsf{hd}(X, Y)$ if $\mathsf{hd}(X, Y) \leq k$ and $k + 1$ otherwise.

A basic property of the Hamming distance is that it is additive:

Observation 9.2.1. *Let U, U', V, V' be strings such that $|U| = |U'|$ and $|V| = |V'|$. We have $\mathsf{hd}(UV, U'V') = \mathsf{hd}(U, U') + \mathsf{hd}(V, V')$.*

A direct consequence is that the Hamming distance cannot increase when removing the first (or last) character of both strings.

Corollary 9.2.2. *For any strings U, V and characters a, b, we have*

$$\mathsf{hd}(U, V) \leq \mathsf{hd}(aU, bV) \text{ and } \mathsf{hd}(U, V) \leq \mathsf{hd}(Ua, Vb).$$

Due to the self-similarity of palindromes and squares, the Hamming distance from a string U to PAL and SQ can be measured in terms of the self-similarity of U.

Property 9.2.3. *Let U be a string of length m, and let $U_1 = U[..\lfloor m/2 \rfloor]$ and $U_2 = U(\lceil m/2 \rceil ..]$. We have*

$$\mathsf{hd}(U, \mathrm{PAL}) = \mathsf{hd}(U_1, U_2^R) = \frac{1}{2}\mathsf{hd}(U, U^R).$$

Proof. We show the first equality via two inequalities. Let P denote the palindrome $U_2^R U_2$ if m is even and $U_2^R U[\lceil m/2 \rceil] U_2$ otherwise. The Hamming distance between U and P is $\mathsf{hd}(U_1, U_2^R)$, hence we have $\mathsf{hd}(U, \mathrm{PAL}) \leq \mathsf{hd}(U_1, U_2^R)$.

Conversely, let V be a palindrome such that $\mathsf{hd}(U, V) = \mathsf{hd}(U, \mathrm{PAL})$. We similarly decompose V into $V_1 V_1^R$ (or $V_1 b V_1^R$ for odd m) and obtain $\mathsf{hd}(U, V) \geq \mathsf{hd}(U_1, V_1) + \mathsf{hd}(U_2, V_1^R)$. Using the fact that $\mathsf{hd}(U_2, V_1^R) = \mathsf{hd}(U_2^R, V_1)$ and applying the triangle inequality, we get $\mathsf{hd}(U_1, U_2^R) \leq \mathsf{hd}(U, V) = \mathsf{hd}(U, \mathrm{PAL})$.

For the second equality, note that $\mathsf{hd}(U_2, U_1^R) = \mathsf{hd}(U_1, U_2^R)$, hence we have

$$\mathsf{hd}(U, U^R) = \mathsf{hd}(U_1, U_2^R) + \mathsf{hd}(U_2, U_1^R) = 2 \cdot \mathsf{hd}(U_1, U_2^R).$$

□

Property 9.2.4. *Each string* $U \in \Sigma^m$ *satisfies* $\mathsf{hd}(U, \mathrm{SQ}) = \mathsf{hd}(U[..m/2], U(m/2..])$ *if* m *is even and* $\mathsf{hd}(U, \mathrm{SQ}) = \infty$ *if* m *is odd.*

Proof. Every square has even length; hence, if m is odd, the distance between U and SQ is infinite. In what follows, we assume that $m = 2i$ for some $i \in \mathbb{N}$. Let $U_1 = U[..i]$ and $U_2 = U(i..]$. By modifying the copy of U_1 in U into U_2, we obtain a square U_2U_2; hence, $\mathsf{hd}(U, \mathrm{SQ}) \leq \mathsf{hd}(U_1, U_2)$.

For the converse inequality, let V^2 be a square such that $\mathsf{hd}(U, \mathrm{SQ}) = \mathsf{hd}(U, V^2)$. We have $|V| = |U_1| = |U_2|$; hence, $\mathsf{hd}(U, V^2) = \mathsf{hd}(U_1, V) + \mathsf{hd}(V, U_2)$. Applying the triangle inequality, we obtain $\mathsf{hd}(U, \mathrm{SQ}) = \mathsf{hd}(U, V^2) \geq \mathsf{hd}(U_1, U_2)$. □

9.2.2 Edit distance, palindromes, and squares

The *edit distance* between two strings U and V, denoted by $\mathsf{ed}(U, V)$, is the minimum number of character insertions, deletions, and substitutions required to transform U into V. For a formal definition, we first rely on the notion of an *alignment* between fragments of strings.

Definition 9.2.5 ([207])**.** *A sequence* $\mathcal{A} = (u_t, v_t)_{i=0}^{m}$ *is an* alignment *of* U *onto* V *if* $(u_0, v_0) = (0, 0)$, $(u_t, v_t) \in \{(u_{t-1} + 1, v_{t-1} + 1), (u_{t-1} + 1, v_{t-1}), (u_{t-1}, v_{t-1} + 1)\}$ *for* $i \in [1..m]$, *and* $(u_m, v_m) = (|U|, |V|)$.

- *If* $(u_t, v_t) = (u_{t-1} + 1, v_{t-1})$, *we say that* $\mathcal{A}$ deletes $U[u_t]$,
- *If* $(u_t, v_t) = (u_{t-1}, v_{t-1} + 1)$, *we say that* $\mathcal{A}$ inserts $V[v_t]$,
- *If* $(u_t, v_t) = (u_{t-1} + 1, v_{t-1} + 1)$, *we say that* $\mathcal{A}$ aligns $U[u_t]$ *and* $V[v_t]$. *If additionally* $U[u_t] = V[v_t]$, *we say that* $\mathcal{A}$ matches $U[u_t]$ *and* $V[v_t]$; *otherwise,* $\mathcal{A}$ substitutes $V[v_t]$ *for* $U[u_t]$.

The *cost* of an alignment $\mathcal{A}$ of U onto V, is the total number of characters that $\mathcal{A}$ inserts, deletes, or substitutes. Now, we define the edit distance $\mathsf{ed}(U, V)$ as the minimum cost of an alignment of U onto V. An alignment of U onto V is *optimal* if its cost is equal to $\mathsf{ed}(U, V)$.

A sequence of edits that an alignment $\mathcal{A}$ uses to transform U into V (specifying the involved positions and characters) is called an *edit sequence* (of the alignment).

Example 9.2.6. A string $U = \mathtt{ababc}$ can be transformed onto $V = \mathtt{bbac}$ by substituting $U[1] = \mathtt{a}$ for $V[1] = \mathtt{b}$ and deleting $U[4] = \mathtt{b}$. The corresponding alignment is $(0, 0)$, $(1, 1)$, $(2, 2)$, $(3, 3)$, $(4, 3)$, $(5, 4)$.

For an integer k, we denote

$$\mathsf{ed}_{\leq k}(X, Y) = \begin{cases} \mathsf{ed}(X, Y) & \text{if } \mathsf{ed}(X, Y) \leq k, \\ k + 1 & \text{otherwise.} \end{cases}$$

For strings $P, T \in \Sigma^*$, we say that a position i is a *k-error occurrence* of P in T if $\mathsf{ed}(T(j..i], P) \leq k$ for some $j \in [1..i]$.

Similar to the Hamming distance, the edit distance cannot increase when removing the first (or last) character of both strings.

Observation 9.2.7. *For any strings* U, V *and characters* a, b, *we have*

$$\mathsf{ed}(U, V) \leq \mathsf{ed}(aU, bV) \text{ and } \mathsf{ed}(U, V) \leq \mathsf{ed}(Ua, Vb).$$

Similarly to the Hamming distance, we can measure the edit distance from a string U to PAL and SQ in terms of the self-similarity of U.

Property 9.2.8. *For any string $U \in \Sigma^*$, we have*

$$\mathsf{ed}(U, \mathrm{PAL}) = \min_i \{\min\{\mathsf{ed}(U[..i], U(i..]^R), \mathsf{ed}(U[..i], U(i+1..]^R)\}\}.$$

As a corollary, $\mathsf{ed}(U, \mathrm{PAL}) = \frac{1}{2}\mathsf{ed}(U, U^R)$.

Proof. First, consider an optimum alignment $(u_t, u'_t)_{t=0}^m$ of U onto U^R. For every $t \in [0..m]$, the alignment maps $U[..u_t]$ onto $U^R[..u'_t]$ and $U(u_t..]$ onto $U^R(u'_t..]$. In particular,

$$\mathsf{ed}(U, U^R) = \mathsf{ed}(U[..u_t], U^R[..u'_t]) + \mathsf{ed}(U(u_t..], U^R(u'_t..]).$$

Suppose that there exists an index $t \in [0..m]$ such that $u_t + u'_t = |U|$. In this case, $U^R[..u'_t] = U(u_t..]^R$ and $U^R(u'_t..] = U[..u_t]^R$. Consequently,

$$\begin{aligned}\mathsf{ed}(U, U^R) &= \mathsf{ed}(U[..u_t], U^R[..u'_t]) + \mathsf{ed}(U(u_t..], U^R(u'_t..]) \\ &= \mathsf{ed}(U[..u_t], U(u_t..]^R) + \mathsf{ed}(U(u_t..], U[..u_t]^R) \\ &= 2 \cdot \mathsf{ed}(U[..u_t], U(u_t..]^R),\end{aligned}$$

that is, $\mathsf{ed}(U, U^R) = \mathsf{ed}(U[..i], U(i..]^R)$ holds for $i = u_t$.

Otherwise, as the sequence $(u_t + u'_t)_{t=0}^m$ increases from 0 to $2|U|$, there exists $t \in [1..m]$ such that $u_{t-1} + u'_{t-1} < |U| < u_t + u'_t$. In particular, $(u_t, u'_t) = (u_{t-1} + 1, u'_{t-1} + 1)$, so $U^R[..u'_{t-1}] = U(u_t..]^R$, $U^R(u'_{t-1}..u'_t] = U(u_{t-1}..u_t]$, and $U^R(u'_t..] = U[..u_{t-1}]$. Consequently,

$$\begin{aligned}\mathsf{ed}(U, U^R) &= \mathsf{ed}(U[..u_{t-1}], U^R[..u'_{t-1}]) \\ &\quad + \mathsf{ed}(U(u_{t-1}..u_t], U^R(u'_{t-1}..u'_t]) \\ &\quad + \mathsf{ed}(U(u_t..], U^R(u'_t..]) \\ &= \mathsf{ed}(U[..u_{t-1}], U(u_t..]^R) \\ &\quad + \mathsf{ed}(U(u_{t-1}..u_t], U(u_{t-1}..u_t]^R) \\ &\quad + \mathsf{ed}(U(u_t..], U[..u_{t-1}]^R) \\ &= 2 \cdot \mathsf{ed}(U[..u_{t-1}], U(u_t..]^R),\end{aligned}$$

that is, $\mathsf{ed}(U, U^R) = \mathsf{ed}(U[..i], U(i+1..]^R)$ holds for $i = u_{t-1} = u_t - 1$.

This completes the proof that

$$2 \min_i \{\min\{\mathsf{ed}(U[..i], U(i..]^R), \mathsf{ed}(U[..i], U(i+1..]^R)\}\} \leq \mathsf{ed}(U, U^R) \tag{9.1}$$

Next, consider a palindrome V such that $\mathsf{ed}(U, \mathrm{PAL}) = \mathsf{ed}(U, V)$. The triangle inequality implies

$$\begin{aligned}\mathsf{ed}(U, U^R) &\leq \mathsf{ed}(U, V) + \mathsf{ed}(V, U^R) \\ &= \mathsf{ed}(U, V) + \mathsf{ed}(V^R, U^R) \\ &= 2\mathsf{ed}(U, V) = 2\mathsf{ed}(U, \mathrm{PAL}).\end{aligned} \tag{9.2}$$

Finally, for any palindrome V, we have $\mathsf{ed}(U, \mathrm{PAL}) \leq \mathsf{ed}(U, V)$. In what follows, we give for each $i \in [1..|U|]$ palindromes V_i, V_i' such that $\mathsf{ed}(U, V_i) \leq \mathsf{ed}(U[..i], U(i..]^R)$ and $\mathsf{ed}(U, V_i') \leq \mathsf{ed}(U[..i], U(i+1..]^R)$.

Let $V_i = U(i..]^R \cdot U(i..]$: it is a palindrome, and we can obtain it from U by transforming its prefix $U[..i]$ into $U(i..]^R$, hence $\mathsf{ed}(U, V_i) \leq \mathsf{ed}(U[..i], U(i..]^R)$. Similarly, $V_i' = U(i+1..]^R \cdot U[i+1] \cdot U(i+1..]$ has the desired property. As a consequence, we obtain

$$\mathsf{ed}(U, \mathrm{PAL}) \leq \min_i \{\min\{\mathsf{ed}(U[..i], U(i..]^R), \mathsf{ed}(U[..i], U(i+1..]^R)\}\}. \tag{9.3}$$

Combining (9.1), (9.2), and (9.3) yields the desired result. □

Corollary 9.2.9. *Let U be a string of length n and $m = \lfloor n/2 \rfloor$. We have*

$$\mathsf{ed}_{\leq k}(U, \mathrm{PAL}) = \min_{i \in [m-k..m+k]} \{\min\{\mathsf{ed}_{\leq k}(U[..i], U(i..]^R), \mathsf{ed}_{\leq k}(U[..i], U(i+1..]^R)\}.$$

Proof. The edit distance between two strings is at least the difference of their lengths, hence we only need to consider strings that differ by at most k in length. □

Property 9.2.10. *Let $U \in \Sigma^n$. We have $\mathsf{ed}(U, \mathrm{SQ}) = \min_i \{\mathsf{ed}(U[..i], U[i+1..])\}$.*

Proof. First, for any $i \in [1..n]$, we have $\mathsf{ed}(U, \mathrm{SQ}) \leq \mathsf{ed}(U[..i], U(i..])$, as editing the substring of U equal to $U[..i]$ into $U(i..]$ yields the string $U(i..]U(i..]$, which is a square.

Now, let V^2 be a square such that $\mathsf{ed}(U, \mathrm{SQ}) = \mathsf{ed}(U, V^2)$. Let r be the position of the rightmost character of U that was not deleted, and whose position in V^2 (after applying the edits) is at most $|V|$. If there is no such character, we set $r = 0$.

We then have $\mathsf{ed}(U, V^2) = \mathsf{ed}(U[..r], V) + \mathsf{ed}(U(r..], V)$. Applying the triangle inequality, we obtain $\mathsf{ed}(U, \mathrm{SQ}) = \mathsf{ed}(U, V^2) \geq \mathsf{ed}(U[..r], U(r..])$, hence we have

$$\mathsf{ed}(U, \mathrm{SQ}) \geq \min_{i \in [1..n]} \{\mathsf{ed}(U[..i], U(i..])\}.$$ □

Corollary 9.2.11. *Let $U \in \Sigma^n$ and $m = \lfloor n/2 \rfloor$. We have*

$$\mathsf{ed}_{\leq k}(U, \mathrm{SQ}) = \min_{i \in [m-k..m+k]} \min \mathsf{ed}_{\leq k}(U[..i], U(i..]).$$

9.2.3 Models of computation

In this work, we focus on two by now classical models of computation: streaming and read-only random access.

In the streaming model, we assume that the input string T arrives as a stream, one character at a time. We account for all the space used, including the space needed to store any information about T. In contrast, in the read-only model, we assume that, after receiving $T[i]$, we have read-only random access to all of $T[1..i]$, and measure only the additional space required for the computation.

In both models, we report the results *online*, i.e. for each prefix $T[1..i]$, we must report the distance to PAL or SQ after receiving $T[i]$ and before receiving $T[i+1]$.

9.3 Streaming algorithms

In this section, we give streaming algorithms for all six problems presented in the introduction.

Our solutions rely on space-efficient *distance sketches*, an extension of Karp-Rabin fingerprints. The Karp-Rabin fingerprints [190] are a family of hash functions over strings that can be updated when a character is appended to the string without recomputing the hash from scratch. Furthermore, if two strings are different, then their hashes are different with high probability. In summary, these functions allow testing whether two strings are equal, i.e. whether the distance between them is 0 or greater than 0.

The distance sketches described in this section are a generalization of Karp-Rabin fingerprints, which were first introduced to solve approximate pattern matching in streaming [61, 107]. Distance sketches allow testing whether the distance between two strings is at most k, and if so, computing that distance. We give an overview of Hamming distance sketches in Section 9.3.1.1 and edit distance sketches in Section 9.3.2.1.

9.3.1 Streaming algorithm for Hamming distance to PAL

9.3.1.1 Hamming distance sketches

We use the Hamming distance sketches introduced by Clifford et al. [107] to solve the streaming k-mismatch problem. Their construction is the following:

Fact 9.3.1. *There exists a function* $\mathsf{sk}_k^{\mathsf{hd}}$ *(parameterized by a constant* $c > 1$*, integers* $n \geq k \geq 1$*, and a seed of* $O(\log n)$ *random bits) that assigns an* $O(k \log n)$*-bit sketch to each string in* $\Sigma^{\leq n}$*. Moreover:*

1. *There is an* $O(k \log^2 n)$*-time* encoding *algorithm that given* $U \in \Sigma^{\leq k}$*, builds* $\mathsf{sk}_k^{\mathsf{hd}}(U)$*.*
2. *There is an* $O(k \log n)$*-time algorithm that, given any two among* $\mathsf{sk}_k^{\mathsf{hd}}(U), \mathsf{sk}_k^{\mathsf{hd}}(V)$*, or* $\mathsf{sk}_k^{\mathsf{hd}}(UV)$*, computes the third one (provided that* $|UV| \leq n$*).*
3. *There is an* $O(k \log^3 n)$*-time* decoding *algorithm that, given* $\mathsf{sk}_k^{\mathsf{hd}}(U)$ *and* $\mathsf{sk}_k^{\mathsf{hd}}(V)$*, computes* $\mathsf{MI}(U, V)$ *if* $\mathsf{hd}(U, V) \leq k$ *and otherwise reports that* $\mathsf{hd}(U, V) > k$*. The error probability is* $O(n^{-c})$*.*

9.3.1.2 Algorithm

We now show that the sketches described in Fact 9.3.1 give a simple algorithm for k-LHD-PAL, improving upon the result of Amir and Porat [31] and achieving the time complexity of $\tilde{O}(k)$ per character.

Theorem 9.3.2. *There is a randomised streaming algorithm that solves the* k-LHD-PAL *problem using* $O(k \log n)$ *bits of space and* $O(k \log^3 n)$ *time per character. The algorithm errs with probability inverse-polynomial in* n*.*

Proof. Using Property 9.2.3, we can reduce the k-LHD-PAL problem to that of computing the threshold Hamming distance between the current prefix of the input string and its reverse. The algorithm maintains the sketches $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[.\,.i])$ and $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[.\,.i]^R)$. When it receives $T[i]$, it constructs $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[i])$, updates both $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[.\,.i])$ and $\mathsf{sk}_{2k}^{\mathsf{hd}}(T[.\,.i]^R)$, and computes $d = \mathsf{hd}_{\leq 2k}(T[.\,.i], T[.\,.i]^R)$ (in $O(k \log^3 n)$ total time, by Fact 9.3.1). Property 9.2.3 implies $\mathsf{hd}_{\leq k}(T[.\,.i], \mathrm{PAL}) = d/2$. The error probability of the algorithm follows from the error probability for the decoding algorithm for Hamming distance sketches. □

The algorithm uses $O(k \log n)$ bits of space, which is nearly optimal. Indeed, by Property 9.2.3, if $U = VW$, with $|V| = |W|$, then $\mathsf{hd}(U, U^R) = 2 \cdot \mathsf{hd}(V, W^R)$. Furthermore, there is a lower bound of $\Omega(k)$ bits on the communication complexity of computing the Hamming distance between two strings V, W [178]. Therefore, using a standard reduction from one-way communication complexity protocols to streaming algorithms , we obtain a lower bound of $\Omega(k)$ bits for the space complexity of streaming algorithms for the k-LHD-PAL problem.

9.3.2 Streaming algorithm for edit distance to PAL

Next, by replacing the Hamming distance sketches by edit distance sketches, we obtain a similar algorithm for k-LED-PAL.

9.3.2.1 Locally consistent decompositions and edit distance sketches

For the edit distance, our algorithms use locally consistent string decompositions and the edit distance sketches of Bhattacharya and Koucký [61]. We store additional information along with the sketches to ensure that we can both prepend and append characters to a sketch.

Locally consistent string decompositions.

The randomized decomposition algorithm of Bhattacharya and Koucký [61][1], which we call the BK-decomposition algorithm, receives two integers k, n as input, and chooses random of hash and compression functions $(H_\ell), (C_\ell)$ (which we discuss later). Then, given a string U of length at most n as input, the algorithm outputs a sequence $\mathbb{G}(U)$ of context-free grammars $\mathbb{G}(U) = G_1^U \cdots G_s^U$ with certain properties, called run-length straight-line programs, or RLSLP for short[2] (the exact definition of an RLSLP is not important for this work; we refer the interested reader to [246] for more details). We denote the length s of the sequence by $|\mathbb{G}(U)|$ and extend notation for indexing and concatenating strings (e.g. sequences of characters) to sequences of grammars in a natural way. Each grammar $\mathbb{G}(U)[i]$ output by the algorithm represents a *unique* string, denoted by $\mathsf{eval}(\mathbb{G}(U)[i])$. We homomorphically extend eval over grammar sequences: for grammars $G_1, \ldots, G_s$, we define $\mathsf{eval}(G_1 \cdots G_s) = \mathsf{eval}(G_1) \cdots \mathsf{eval}(G_s)$.

The BK-decomposition satisfies the following properties:

Fact 9.3.3 ([62, Theorem 3.1]). *Let U, V be a pair of strings of length at most n such that $\mathsf{ed}(U, V) \leq k$. Let $\mathbb{G}(U)$ and $\mathbb{G}(V)$ be the sequences of grammars output by the BK-decomposition algorithm on inputs U and V respectively using the same choice of random hash and compression functions. The following is true for n large enough:*

1. *With probability at least $1 - 2/n$, $U = \mathsf{eval}(\mathbb{G}(U))$ and $V = \mathsf{eval}(\mathbb{G}(V))$;*
2. *With probability at least $1 - 2/\sqrt{n}$, for all i, j, the grammars $\mathbb{G}(U)[i]$ and $\mathbb{G}(V)[j]$ have size $\tilde{O}(k)$;*

[1] In what follows, we will refer to the full version [62] of [61], as we will need to adapt proofs that only appear in the full version.

[2] While it is not stated explicitly in [62], the grammars returned by their algorithm are RLSLPs, up to adding rules of the form $S_a \to a$ for every $a \in \Sigma$ and applying the algorithm to the word obtained by replacing a by S_a.

3. *With probability at least* 0.9*, there is an integer* $s = |\mathbb{G}(U)| = |\mathbb{G}(V)|$*, we have* $\mathbb{G}(U)[i] = \mathbb{G}(V)[i]$ *for all but at most* k *indices* i *such that* $1 \leq i \leq s$*, and*

$$\mathsf{ed}(U, V) = \sum_{i=1}^{s} \mathsf{ed}(\mathsf{eval}(\mathbb{G}(V)[i]), \mathsf{eval}(\mathbb{G}(U)[i])).$$

Overview of the decomposition algorithm. The BK-decomposition algorithm processes the input string U in $\lambda = O(\log n)$ stages[3], gradually compressing the string. The algorithm uses a work alphabet Γ that contains Σ and has size polynomial in n. The compression process iteratively replaces pairs of characters $(a, b) \in \Gamma^2$ by another character $c_{ab} \in \Gamma$, in a lossless manner. (The BK-decomposition algorithm is probabilistic, the description that we give here is true with high probability, e.g. the compression is lossless with probability at least $1 - 1/\operatorname{poly}(n)$. In what follows, we assume that this property holds.) The goal of the BK-decomposition algorithm is to obtain similar output sequences for similar input strings, so characters in identical substrings should be compressed the same way. To obtain this property, Bhattacharya and Koucký [62] use *locally consistent coloring* functions to choose which pairs of characters should be replaced and compressed. These coloring functions have the following properties:

Fact 9.3.4 ([62, Proposition 2.4])**.** *There exists a function* $F_{CVL} : \Gamma^* \to \{1, 2, 3\}^*$ *with the following properties. Let* $\rho = O(\log^* n)$*. For each string* $U \in \Gamma^*$ *with no two identical consecutive characters:*

1. $|F_{CVL}(U)| = |U|$ *and* $F_{CVL}(U)$ *can be computed in time* $O(\rho|U|)$*,*
2. *for each* $i = 1, \ldots, |U|$*, the* i*-th character of* $F_{CVL}(U)$ *is a function of* $U[i - \rho \,..\, i + \rho]$ *only,*
3. *no two consecutive characters of* $F_{CVL}(U)$ *are the same,*
4. *out of every three consecutive characters of* $F_{CVL}(U)$*, at least on of them is* 1*.*

To choose which symbols to compress, the BK-decomposition algorithm computes $F_{CVL}(U)$, and replaces $U[i]U[i+1]$ with $C_\ell(U[i]U[i+1])$ at every position i such that the coloring is 1. Here, $(C_\ell)_{0,\ldots,\lambda}$ are a family of random functions $\Gamma^2 \to \Gamma$. By taking Γ sufficiently large, the compression is lossless with high probability. The ℓ-th function C_ℓ is used for the ℓ-th compression iteration. Item 3 ensures that the compression process is well defined, i.e. if the i-th position is colored 1, the $i+1$-th cannot be. Item 4 ensures that the length of the compressed string is at most two thirds of the length of U, hence after $\lambda = O(\log n)$ steps, the string is reduced to to a single character. A string U can only be colored with F_{CVL} if it does not contain two identical consecutive characters. To ensure that this holds, U is run-length encoded before running the compression process, i.e. maximal runs of the form a^r for $a \in \Gamma$ and $r \geq 2$ are replaced by a special character $c_{a,r} \in \Gamma$ that encodes the character and the length of the run, followed by a special character #.

We will later talk about the *decompression process*: it corresponds to the inverse operation. To compute the decompression at level ℓ, every character c in U such that there exist $a, b \in \Gamma$ such that $C_\ell(a, b) = c$ is replaced by ab.

The other important operation of the BK-decomposition algorithm is *splitting* strings into blocks (i.e. substrings). A string U is split into blocks using a family of hash functions $H_\ell : \Gamma^2 \to \{0, \ldots, D\}$ for $D = O(k \log n \log^* n)$ and $\ell = 0, \ldots, \lambda$. At stage ℓ, the string U

[3] In [62], λ is denoted by L, τ by T, ρ by R, and μ by M. We changed the notation to avoid collisions.

is split at position i when $H_\ell(U[i], U[i+1]) = 0$. The family of hash functions is chosen so that this happens with probability $1/D$ for any symbols $U[i], U[i+1]$.

The BK-decomposition algorithm repeatedly applies these operations. If a string has length 1, we convert it into a grammar and append it to the decomposition (the algorithm for constructing the grammar is beyond the scope of this chapter). Otherwise, the string is compressed, the resulting string is then split, and the procedure is applied recursively to each block obtained from splitting. The index ℓ of hash and compression functions used corresponds to the recursion depth.

Dynamically updating the BK-decomposition. Bhattacharya and Koucký [62] show that this decomposition can be updated efficiently when appending a character to a string. In what follows, we argue that the same is true for *prepending* a character. While it may happen that $\mathbb{G}(UV)$ is shorter than $\mathbb{G}(U)$, these decompositions are related as follows:

Corollary 9.3.5 (Of [62, Lemmas 4.1, 4.2])**.** *Consider $U, V \in \Sigma^*$ such that $|U| + |V| \le n$ and let $\tau = \lambda \cdot \rho = O(\log n \log^* n)$. Let $G = \mathbb{G}(U)$, $G' = \mathbb{G}(UV)$, $G'' = \mathbb{G}(VU)$, and $s = |G|$, $s' = |G'|$, $s'' = |G''|$. We have:*

1. *$G[..s-\tau] = G'[..s-\tau]$ and $|U| \le |\mathsf{eval}(G'[..\min\{s+\tau, s'\}])|$,*
2. *$G(\tau..] = G''(s''-s+\tau..]$ and $|U| \le |\mathsf{eval}(G''(\max\{s''-s+\tau, 0\}..])|$.*

Proof. Item 1 is a direct consequence of [62, Lemmas 4.1, 4.2].

Item 2 can be proved using similar ideas, which we sketch here. Consider the coloring $X = F_{CVL}(U)$ and $Y = F_{CVL}(VU)$ obtained using the locally consistent coloring of Fact 9.3.4. Because of the local property of F_{CVL} (Item 2 in Fact 9.3.4), we have $X[\rho..] = Y[|V| + \rho..]$, therefore the compression and the splitting are the same in the last $|U| - \rho$ characters of VU and of U. This is for the first compression level. Similarly, at each level, the result may only change for ρ additional characters of U, and there are a total of $\lambda = O(\log n)$ levels. Therefore, overall, the part that corresponds to U in both grammar sequences differs by at most $\tau = O(\log n \log^* n)$ grammars. □

In particular, this result implies that if the index of a grammar in $\mathbb{G}(U)$ is less than $|\mathbb{G}(U)| - \tau$ (resp. more than τ), then appending (resp. prepending) characters to U will not change that grammar. Extending the terminology of [62], we call a grammar that remains unchanged when appending (resp. prepending) any string a *right-committed grammar* (resp. *left-committed grammar*), while the others are *right-active* (resp. *left-active*) *grammars*. When a grammar is both left- and right-committed, we simply say that it is *committed*. In what follows, we use $\alpha_r(\mathbb{G}(U))$ (resp. $\alpha_l(\mathbb{G}(U))$) to denote the number of right-active (resp. left-active) grammars in $\mathbb{G}(U)$. Corollary 9.3.5 implies that $\alpha_l(\mathbb{G}(U)), \alpha_r(\mathbb{G}(U)) \le \tau$.

Theorem 5.1 in [62] presents an algorithm only for the case of appending a character, we argue that essentially the same can be used to prepend one.

Corollary 9.3.6 (Of [62, Theorem 5.1])**.** *Let $U \in \Sigma^{\le n-1}$, $G = \mathbb{G}(U)$, and $s = |G|$. There are two algorithms* AppendCharacter *and* PrependCharacter *that run in $\tilde{O}(k)$ time and:*

1. *Given a character $a \in \Sigma$ and $G[\max\{1, s-\tau\}..s]$, the algorithm* AppendCharacter *outputs G' such that $\mathbb{G}(Ua) = G[1..\max\{1, s-\tau\}]G'$ and $|G'| \le 4\tau\lambda$.*
2. *Given a character $a \in \Sigma$ and $G[1..\min\{\tau+1, s\}]$, the algorithm* PrependCharacter *outputs G' such that $\mathbb{G}(aU) = G' \cdot G(\min\{\tau+1, s\}..s]$ and $|G'| \le 4\tau\lambda$.*

Proof. Bhattacharya and Koucký [62, Theorem 5.1] shows how to append a character to a grammar decomposition. The algorithm for appending a character a starts from the string Z obtained by concatenating the starting symbol of each grammar in G. It then decompresses the end of Z, until the decompression has length at least τ, appends a, and recompresses the result. The algorithm of Bhattacharya and Koucký [62] uses several subroutines (Algorithms 5-15 from [62, Section 5]) to compute which part to decompress, and to ensure efficient recompression.

We now argue that this algorithm can be adapted to prepend a character. Algorithms 5, 6, 7, 8 and 11 of [62] are not specific to appending a character, so they do not need to be adapted. Algorithm 9 finds the shortest prefix of a compressed string Z whose decompression at level ℓ has a length exceeding a given threshold: it can easily be adapted to find the shortest suffix with the same property by starting from the end of Z. Algorithm 10 partially decompresses a string Z, starting from the end, until it reaches length τ: by iterating on Z from left to right, we obtain the symmetric version which decompresses starting from the beginning. Algorithms 12-15 use Algorithms 5-11 as subroutines. Their symmetric versions can be obtained by working symmetrically and using the symmetric versions of Algorithms 9 and 10 described above. □

It follows from the above two results that, by storing the last $\alpha_r(\mathbb{G}(U))+\tau$ (resp. first $\alpha_l(\mathbb{G}(U))+\tau$) grammars of $\mathbb{G}(U)$, we can compute the last $\alpha_r(\mathbb{G}(UV))+\tau$ grammars of $\mathbb{G}(UV)$ (resp. first $\alpha_l(\mathbb{G}(VU))+\tau$ grammars of $\mathbb{G}(VU)$) by applying the corresponding algorithm of Corollary 9.3.6 $|V|$ times. These grammars include the right-active grammars of $\mathbb{G}(UV)$ (resp. left-active grammars of $\mathbb{G}(VU)$).

Edit distance sketches.

Our algorithms exploit an extension of the *edit distance sketches* of Bhattacharya and Koucký [62], which are based on the above grammar decomposition.

Fact 9.3.7 ([62, Lemma 3.13]). *Let $\mu = \tilde{O}(k)$ be a parameter (see Footnote 3). There is an injective mapping* enc *from the set of grammars output by the decomposition algorithm to the set of strings of length μ on an alphabet of size polynomial in n that guarantees that the following is satisfied:*

1. *A grammar can be encoded and decoded in $O(\mu)$ time;*
2. *The encodings of two equal grammars are equal;*
3. *The encodings of two distinct grammars output by the decomposition algorithm differ in all μ characters with probability at least $1 - 2\mu/n$.*

We homomorphically extend this encoding to sequences of grammars: $\mathsf{enc}(G_1 \cdots G_s) = \mathsf{enc}(G_1) \cdots \mathsf{enc}(G_s)$. We are now ready to describe our extension of the edit distance sketches of [62]. Informally, to construct the sketch of a string U, we first compute its grammar decomposition $\mathbb{G}(U)$, and compute its encoding $E_U = \mathsf{enc}(\mathbb{G}(U))$. The sketch $\mathsf{sk}^{\mathsf{ed}}_k(U)$ is given by the *Hamming distance* sketch of E_U with threshold $k' = k \cdot \mu$. If the edit distance between two strings U and V is at most k, then by Fact 9.3.3, their grammar decompositions differ in at most k grammars, and in turn E_U and E_V differ in at most $k' = k \cdot \mu$ positions, by definition of enc. Recall from Fact 9.3.1 that, in this case, given $\mathsf{sk}^{\mathsf{hd}}_{k'}(E_U)$ and $\mathsf{sk}^{\mathsf{hd}}_{k'}(E_V)$, we can compute the mismatch information between E_U and E_V, that is, the list of positions where these strings differ and the characters at those positions. By Fact 9.3.7, we have a high probability of recovering the encoding of all grammars that differ, from which we can recover the differing grammars themselves,

which finally allows computing the edit distance. The time and space complexity of these operations is $\tilde{O}(k') = \tilde{O}(k^2)$. The actual sketches are slightly more complex: we only encode in E_U the committed grammars, and explicitly store a small superset of active grammars.

In the remainder of this section, we give a formal and detailed construction of the edit distance sketches, building on the work of Bhattacharya and Koucký [62]. Our result is the following.

Lemma 9.3.8. *There is an edit distance sketch $\mathsf{sk}_k^{\mathsf{ed}}$ that uses $\tilde{O}(k^2)$ bits of space, and supports the following operations assuming that all involved strings have length at most n:*

1. ***Append:*** *given $\mathsf{sk}_k^{\mathsf{ed}}(U)$ and a character $a \in \Sigma$, compute $\mathsf{sk}_k^{\mathsf{ed}}(Ua)$,*
2. ***Prepend:*** *given $\mathsf{sk}_k^{\mathsf{ed}}(U)$ and a character $a \in \Sigma$, compute $\mathsf{sk}_k^{\mathsf{ed}}(aU)$,*
3. ***Distance:*** *given $\mathsf{sk}_k^{\mathsf{ed}}(U)$ and $\mathsf{sk}_k^{\mathsf{ed}}(V)$, compute $\mathsf{ed}_{\le k}(U, V)$.*

All three operations take $\tilde{O}(k^2)$ time and space and err with probability inverse polynomial in n.

Proof. Let $\mathbb{G}(U)$ be the locally consistent decomposition of a string U. In what follows, we denote $\alpha_r = \alpha_r(\mathbb{G}(U))$ and $\alpha_l = \alpha_l(\mathbb{G}(U))$. If $s = |\mathbb{G}(U)| \le 4\tau$, $\mathsf{sk}_k^{\mathsf{ed}}(U) = \mathsf{enc}(\mathbb{G}(U))$ and takes $\tilde{O}(k)$ space. Otherwise, $\mathsf{sk}_k^{\mathsf{ed}}(U)$ is defined as a tuple

$$(\mathbb{G}_l, \mathcal{S}, \mathbb{G}_r) = \left(\mathbb{G}(U)[1..s_l], \mathsf{sk}_{k\cdot\mu}^{\mathsf{hd}}(\mathsf{enc}(\mathbb{G}(U)(s_l..s - s_r])), \mathbb{G}(U)(s - s_r..s]\right),$$

where s_l and s_r are integers satisfying the invariant

$$2\tau \ge s_l \ge \alpha_l + \tau \text{ and } 2\tau \ge s_r \ge \alpha_r + \tau. \tag{9.4}$$

Note that this definition does not define $\mathsf{sk}_k^{\mathsf{ed}}(U)$ uniquely (as there are multiple possible choices of s_l and s_r), but the invariant of Eq. (9.4) ensures that we can always perform updates and distance operations using the sketch. Initially (i.e., the first time s becomes larger than 4τ), we set $s_l = s_r = 2\tau$, which satisfy the invariant of Eq. (9.4) by Corollary 9.3.5.

Update operations. Let $(\mathbb{G}_l, \mathcal{S}, \mathbb{G}_r)$ be the sketch of U, $s_l = |\mathbb{G}_l|$ and $s_r = |\mathbb{G}_r|$. Given a character $a \in \Sigma$, we obtain the sketch of Ua feeding $(\mathbb{G}_r[s_r - \tau..s_r], a)$ into the algorithm AppendCharacter (Corollary 9.3.6). Let G_{out} be the sequence of grammars output by this algorithm and let $G' = \mathbb{G}_r[..s_r - \tau)G_{out}$ be the sequence of grammars obtained by replacing $\mathbb{G}_r[s_r - \tau..s_r]$ with G_{out} in $\mathbb{G}_r$. If $|G'| \le 2\tau$, we output $(\mathbb{G}_l, \mathcal{S}, G')$ for the sketch of Ua. Otherwise, we only store explicitly the last 2τ grammars of G', and add the others at the end of the Hamming sketch $\mathcal{S}$. More formally, we construct the Hamming distance sketch $\mathcal{S}'$ by concatenating $\mathcal{S}$ and $\hat{\mathcal{S}} = \mathsf{sk}_{k\cdot\mu}^{\mathsf{hd}}(\mathsf{enc}(G'[..|G'| - 2\tau]))$. We then output $\mathsf{sk}_k^{\mathsf{ed}}(Ua) = (\mathbb{G}_l, \mathcal{S}', G'(|G'| - 2\tau..])$. By Corollary 9.3.6, $|G'| \le 4\tau\lambda + \tau = \tilde{O}(1)$, hence we can build $\hat{\mathcal{S}}$ in time $\tilde{O}(k^2)$ using Fact 9.3.1, and the subsequent sketch concatenation also takes $\tilde{O}(k^2)$ time. In both cases, Corollary 9.3.5 ensures that the invariant of Eq. (9.4) is satisfied.

Prepending a character works similarly using $\mathbb{G}_l$ and the PrependCharacter algorithm of Corollary 9.3.6.

Distance operation. Let U, V be strings satisfying the conditions of Fact 9.3.3, and consider the sketches $\mathsf{sk}_k^{\mathsf{ed}}(U) = (\mathbb{G}_l, \mathcal{S}, \mathbb{G}_r)$ and $\mathsf{sk}_k^{\mathsf{ed}}(V) = (\mathbb{G}'_l, \mathcal{S}', \mathbb{G}'_r)$ obtained from decompositions such that the properties of Fact 9.3.3 hold. By Fact 9.3.3(3), at most k grammars of the decompositions of U and V differ: we can recover them from $\mathsf{sk}_{k\cdot\mu}^{\mathsf{hd}}(\mathsf{enc}(\mathbb{G}(U)))$

and $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}(V)))$. We can obtain $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}(U)))$ by concatenating $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}_l))$, $\mathcal{S}$ and $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}_r))$, which takes $\tilde{O}(k^2)$ time, and similarly for $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}(V)))$. Notice that the above operation results in $\mathsf{sk}^{\mathsf{hd}}_{k\cdot\mu}(\mathsf{enc}(\mathbb{G}(U)))$ regardless of the length of $|\mathbb{G}_l|$ and $|\mathbb{G}_r|$: this normalization helps us avoid issues related to the definition of the sketches. Using these sketches, we can recover $\mathsf{enc}(G_i)$ for every i such that $\mathbb{G}(U)[i] \neq \mathbb{G}(V)[i]$, and by decoding we recover the corresponding grammar G_i. We can then compute the edit distance between U and V using Fact 9.3.3(3):

$$\begin{aligned}\mathsf{ed}(U,V) &= \sum_{i=1}^{s} \mathsf{ed}(\mathsf{eval}(\mathbb{G}(U)[i]), \mathsf{eval}(\mathbb{G}(V)[i])) \\ &= \sum_{i:\mathbb{G}(U)[i]\neq\mathbb{G}(V)[i]} \mathsf{ed}(\mathsf{eval}(\mathbb{G}(U)[i]), \mathsf{eval}(\mathbb{G}(V)[i]))\end{aligned}$$

Given two RLSLPs G, G' of size at most m and $d = \mathsf{ed}(\mathsf{eval}(G), \mathsf{eval}(G'))$, one can compute $\min\{d, k+1\}$ in time $\tilde{O}(m+k^2)$ using the algorithm of Ganesh et al. [156]. Using binary search over k, we can reduce this running time to $\tilde{O}(m+d^2)$ when $d \leq k$. Therefore, we can compute the above sum in time $\tilde{O}(k^2)$: there are k RLSLPs of size $\tilde{O}(k)$, and we can stop the computation as soon as the sum of $d_i = \mathsf{ed}(\mathsf{eval}(\mathbb{G}(U)[i]), \mathsf{eval}(\mathbb{G}(V)[i]))$ exceeds k. In total, this costs $\tilde{O}(\mu + k^2 + \sum_i(\mu + d_i^2) \leq k\mu + (\sum_i d_i)^2) = \tilde{O}(k^2)$ time. The space complexity can be upper-bounded (up to polylog factors) by the time complexity.

Probability of success. The results of Fact 9.3.3 hold with constant probability, which can be boosted in a standard way by repeating the scheme a logarithmic number of times. □

9.3.2.2 Algorithm

Theorem 9.3.9. *There is a randomised streaming algorithm that solves the k-LED-PAL problem for a string of length n using $\tilde{O}(k^2)$ bits of space and $\tilde{O}(k^2)$ time per character.*

Our algorithm maintains two sketches x, x' initialised to sketches of the empty string $\mathsf{sk}^{\mathsf{ed}}_{2k}(\varepsilon)$. Upon receiving $T[i]$, we append it to x and prepend it to x'. After performing this update, we have $x = \mathsf{sk}^{\mathsf{ed}}_{2k}(T[..i])$, and $x' = \mathsf{sk}^{\mathsf{ed}}_{2k}(T[..i]^R)$. We then use them to compute $\mathsf{ed}_{\leq 2k+1}(T[..i], T[..i]^R)$, which gives us $\mathsf{ed}_{\leq k}(T[..i], \mathrm{PAL})$ by Property 9.2.8. By Lemma 9.3.8, updating the sketches and using them to compute the edit distance takes $\tilde{O}(k^2)$ time per character and $\tilde{O}(k^2)$ bits of space.

9.3.3 Streaming algorithms for approximating the longest approximate subpalindrome

In this section, we use the streaming algorithms of Theorem 9.3.2 and Theorem 9.3.9 together with the framework of Gawrychowski et al. [161] to give algorithms for $(1+\varepsilon)$-k-HD-PAL and $(1+\varepsilon)$-k-ED-PAL.

Gawrychowski et al. [161] introduce a function ttl_ε (which stands for *time to live*) which satisfies the following property: for any interval $[i..j]$, there exists a position r such that there is a prefix of the interval $[r..r+ttl_\varepsilon(r)]$ centered at the same position as $[i..j]$ and has length at least $(j-i+1)/(1+\varepsilon)$.

More formally, the properties of the ttl_ε function can be summarized as follows:

Fact 9.3.10 ([161, Lemmas 5-8]). *For any $0 < \varepsilon \leq 1$, there exists a function $ttl_\varepsilon : \mathbb{N} \to \mathbb{N}$ satisfying each of the following:*

1. *$ttl_\varepsilon(n)$ can be computed in time and space $O(\log(n)/\varepsilon)$,*
2. *for all integers $i < j$, there exists an integer r, $i \leq r \leq j$, such that $r + ttl_\varepsilon(r) \geq j - (r - i)$ and $ttl_\varepsilon(r) \geq (j - i + 1)/(1 + \varepsilon)$,*
3. *for every integer i, $1 \leq i \leq n$, there are $O(\frac{\log(n\varepsilon)}{\varepsilon})$ integers r such that $r \leq i \leq r + ttl_\varepsilon(r)$.*

As a corollary we obtain that if $T[i..j]$ is a palindrome of length ℓ, then there exists r such that $T[r..r + ttl_\varepsilon(r)]$ has a prefix palindrome of length $\ell/(1+\varepsilon)$. The same property holds if we replace "palindrome" with "string within distance k of PAL" in the previous sentence.

This gives a streaming algorithm for solving $(1+\varepsilon)$-k-HD-PAL in T: for every position r, start an instance of the algorithm for k-LHD-PAL on $T[r..r + ttl_\varepsilon(r)]$, and return the length of the longest prefix found over all instances of the algorithm. The next observation follows from Property 9.2.3 and Corollary 9.2.2.

Observation 9.3.11. *Let P be a string of length m such that $\mathsf{hd}(P, \mathrm{PAL}) \leq k$. Then, for any $t \leq \lfloor m/2 \rfloor$, we have $\mathsf{hd}(P[t..m - t + 1], \mathrm{PAL}) \leq k$.*

Corollary 9.3.12. *For any $0 < \varepsilon \leq 1$, there is a streaming algorithm for $(1 + \varepsilon)$-k-HD-PAL that uses $O(\frac{k \log^4 n}{\varepsilon})$ time per character and $O(\frac{k \log^2 n}{\varepsilon})$ bits of space.*

Proof. Correctness follows from the above observation and Observation 9.3.11. Fact 9.3.10 ensures that at most $O(\frac{\log(n\varepsilon)}{\varepsilon})$ copies of the algorithm of Theorem 9.3.2 are running in parallel at every position: the time and space complexities follow. □

To obtain a similar algorithm for $(1 + \varepsilon)$-k-ED-PAL, we give an analogue of Observation 9.3.11 for the edit distance.

Observation 9.3.13. *Let P be a string of length m such that $\mathsf{ed}(P, \mathrm{PAL}) \leq k$. Then, for any $t \leq \lfloor m/2 \rfloor$, we have $\mathsf{ed}(P[t..m - t + 1], \mathrm{PAL}) \leq k$.*

This observation follows from Property 9.2.8 and Observation 9.2.7.

Corollary 9.3.14. *For any $0 < \varepsilon \leq 1$, there is a streaming algorithm for $(1 + \varepsilon)$-k-ED-PAL that uses $\tilde{O}(\frac{k^2}{\varepsilon})$ time per character and $\tilde{O}(\frac{k^2}{\varepsilon})$ bits of space.*

9.3.4 A streaming algorithm for k-LHD-SQ

In this section, we show the following theorem:

Theorem 9.3.15. *There is a randomized streaming algorithm that solves the k-LHD-SQ problem for a string $T \in \Sigma^n$ using $O(k \log^2 n)$ bits of space and $\tilde{O}(k)$ time per character. The algorithm errs with probability inverse-polynomial in n.*

Property 9.2.4 allows us to derive $\mathsf{hd}_{\leq k}(T[..2i], \mathrm{SQ})$ from the sketches $\mathsf{sk}^{\mathsf{hd}}_k(T[..i])$ and $\mathsf{sk}^{\mathsf{hd}}_k(T[..2i])$: we can combine them to obtain $\mathsf{sk}^{\mathsf{hd}}_k(T(i..2i])$, and a distance computation on $\mathsf{sk}^{\mathsf{hd}}_k(T[..i])$ and $\mathsf{sk}^{\mathsf{hd}}_k(T(i..2i])$ returns $\mathsf{hd}_{\leq k}(T[..i], T(i..2i]) = \mathsf{hd}_{\leq k}(T[..2i], \mathrm{SQ})$.

Naively applying this procedure requires storing the sketch $\mathsf{sk}^{\mathsf{hd}}_k(T[..i])$ until the algorithm has read $T[..2i]$, that is, storing $\Theta(n)$ sketches at the same time. To reduce the number of sketches stored, we use a filtering procedure based on the following observation:

Observation 9.3.16. *If* $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$ *and* $\ell \in [1..i]$*, then* $i+\ell$ *is a* k*-mismatch occurrence of* $T[..\ell]$*, that is,* $\mathsf{hd}(T[..\ell], T(i..i+\ell]) \le k$*.*

Example 9.3.17. For $k = 1, \ell = 2$, and $i = 3$, the word $T[..6] = \mathtt{abcacc}$ is a 1-mismatch square (by Property 9.2.4) and the fragment $T(3..5] = \mathtt{ac}$ is a 1-mismatch occurrence of the prefix $T[..2] = \mathtt{ab}$.

Observation 9.3.16 motivates our filtering procedure: if we choose some prefix $P = T[..\ell]$ of the string, we only need to store the sketch of $T[..i]$ for every $i \ge \ell$ such that $i+\ell$ is a k-mismatch occurrence of P. Clifford et al. [107] showed a data structure $\mathcal{S}$ that exploits the structure of such occurrences and stores them using $O(k \log^2 n)$ bits of space while allowing reporting the occurrence at position $i+\ell$ when $T[i+\ell+\Delta]$ is pushed into $\mathcal{S}$ – we say that $\mathcal{S}$ reports the k-mismatch occurrences of P in T with *a fixed delay* Δ. Our algorithm needs to receive the occurrence at position $i+\ell$ when $T[2i]$ is pushed into the stream, i.e. we require $\mathcal{S}$ to report occurrences with a *non-decreasing* delay. In Section 9.3.4.1 we present a modification of the data structure of [107] to allow non-decreasing delays, and in Section 9.3.4.2 we explain how we use it to implement a space-efficient streaming algorithm for k-LHD-SQ.

9.3.4.1 Reporting k-mismatch occurrences with nondecreasing delay.

The algorithm of Clifford et al. [107] reports additional information along with the positions of the k-mismatch occurrences: specifically, it produces the *stream of* k*-mismatch occurrences of* P *in* T, defined as follows.

Definition 9.3.18 ([107, Definition 3.2])**.** *The stream of* k*-mismatch occurrences of a pattern* P *in a text* T *is a sequence* S_P^k *such that* $S_P^k[i] = (i,\ \mathsf{MI}(T(i-|P|..i], P),\ \mathsf{sk}_k^{\mathsf{hd}}(T[..i-|P|]))$ *if* $\mathsf{hd}(P, T(i-|P|..i]) \le k$ *and* $S_P^k[i] = \perp$ *otherwise.*

As explained next, the algorithm of [107] can report the k-mismatch occurrences with a prescribed delay.

Corollary 9.3.19 (of [107])**.** *There is a streaming algorithm that, given a pattern* P *followed by a text* $T \in \Sigma^n$*, reports the* k*-mismatch occurrences of* P *in* T *using* $O(k \log^2 n)$ *bits of space and* $O(\sqrt{k \log^3 n} + \log^4 n)$ *time per character. The algorithm can report each occurrence* i *with no delay (that is, upon receiving* $T[i]$*) or with any prescribed delay* $\Delta = \Theta(|P|)$ *(that is, upon receiving* $T[i+\Delta]$*). For each reported occurrence* i*, the underlying tuple* $S_P^k[i]$ *can be provided on request in* $O(k \log^2 n)$ *time.*

Proof. If no delay is required, we use [107, Theorem 1.2], which reports k-mismatch occurrences of P in T and, upon request, provides the mismatch information $\mathsf{MI}(T(i-|P|..i], P)$; this algorithm uses $O(k \log^2 n)$ bits of space and takes $O(\sqrt{k \log^3 n} + \log^4 n)$ time per character. We also use [107, Fact 4.4] to maintain the sketch $\mathsf{sk}_k^{\mathsf{hd}}(T[..i])$ (reported on request); this algorithm uses $O(k \log n)$ bits of space and takes $O(\log^2 n)$ time per character.

Whenever requested to provide $S_P^k[i]$ for some k-mismatch occurrence i of P in T, we retrieve the mismatch information $\mathsf{MI}(T(i-|P|..i], P)$ (in $O(k)$ time) and the sketch $\mathsf{sk}_k^{\mathsf{hd}}(T[..i])$ (in $O(k \log^2 n)$ time). Combining $\mathsf{sk}_k^{\mathsf{hd}}(P)$ with $\mathsf{MI}(T(i-|P|..i], P)$, we build $\mathsf{sk}_k^{\mathsf{hd}}(T(i-|P|..i])$ (using [107, Lemma 6.4] in $O(k \log^2 n)$ time) and then derive $\mathsf{sk}_k^{\mathsf{hd}}(T[..i-|P|])$ using Fact 9.3.1 (in $O(k \log n)$ time). Processing the request takes $O(k \log^2 n)$ time and $O(k \log^2 n)$ bits of space overall.

If a delay $\Delta = \Theta(|P|)$ is required, our approach depends on whether there exists $p \in [1..k]$ such that $\mathsf{hd}(P[..|P|-p], P(p..|P|]) \leq 2k$ (such p is called a $2k$-period in [107]). This property is tested using a streaming algorithm of [107, Lemma 4.3], which takes $O(k \log n)$ bits of space, $O(\sqrt{k \log n})$ time per character of P, and requires $O(k\sqrt{k \log n})$-time post-processing (performed while reading $T[..k]$). If P satisfies this condition, then we just use [107, Theorem 4.2], whose statement matches that of Corollary 9.3.19.

Otherwise, [107, Observation 4.1] shows that P has at most one k-mismatch occurrence among any k consecutive positions in T. In that case, we use the aforementioned approach to produce the stream S_P^k with no delay and the buffer of [107, Proposition 3.3] to delay the stream by Δ characters. The buffering algorithm takes $O(k \log^2 n)$ bits of space and processes each character $T[i]$ in $O(k \log^2 n + \log^3 n)$ time (if P has k-mismatch occurrences at positions i or $i-\Delta$) or $O(\sqrt{k \log n} + \log^3 n)$ time (otherwise). Since the former case holds for at most two out of every k consecutive positions, we can achieve $O(\sqrt{k \log^3 n} + \log^4 n)$ worst-case time per character by decreasing the delay to $\Delta - k$ and buffering up to k characters of T and up to k elements of S_P^k. While the algorithm processes $T[i + \Delta]$, the latter buffer already contains $S_P^k[i]$, but $O(k)$ time is still needed to output this value (if $S_P^k[i] \neq \perp$). □

The algorithm of Corollary 9.3.19 has a fixed delay Δ, i.e., it outputs $S_P^k[i]$ upon receiving $T[i+\Delta]$. Our application requires a variable delay: we need to access $S_P^k[i+|P|]$ upon reading $T[2i]$, that is, with a delay of $i - |P|$. We present a black-box construction that extends the data structure of Corollary 9.3.19 to support non-decreasing delays Δ_i, $i \in [1..d]$. Naively, one could use the algorithm $\mathcal{A}$ of Corollary 9.3.19 with a fixed delay Δ_1 and buffer the input characters so that $\mathcal{A}$ receives $T[i + \Delta_1]$ only when we actually process $T[i + \Delta_i]$. Unfortunately, this requires storing $T[i + \Delta_1..i + \Delta_i)$, which could take too much space. Thus, we feed $\mathcal{A}$ with $T[1..\Delta_1]$ followed by blank characters $\perp$ (issued at appropriate time steps without the necessity of buffering input characters) so that $\mathcal{A}$ reports k-mismatch occurrences $i \in [1..\Delta_1]$ with prescribed delays. Then, we use another instance of the algorithm of Corollary 9.3.19, with a fixed delay $\Delta_{1+\Delta_1}$, to output k-mismatch occurrences $i \in (\Delta_1..\Delta_1 + \Delta_{1+\Delta_1}]$; we continue this way until the whole interval $[1..d]$ is covered. We formalise this idea in the following lemma.

Lemma 9.3.20. *Let $\Delta_1 \leq \Delta_2 \leq \cdots \leq \Delta_d$ be a non-decreasing sequence of $d = O(|P|)$ integers $\Delta_i = \Theta(|P|)$, represented by an oracle that reports each element Δ_i in constant time.*

There is a streaming algorithm that, given a pattern P followed by a text T, reports the k-mismatch occurrences of P in T using $O(k \log^2 n)$ bits of space and $O(\sqrt{k \log^3 n} + \log^4 n)$ time per character. The algorithm reports each occurrence $i \in [1..d]$ with delay Δ_i, that is, upon receiving $T[i + \Delta_i]$. For each reported occurrence $i \in [1..d]$, the underlying tuple $S_P^k[i]$ can be provided on request in $O(k \log n)$ time.

Proof. We use multiple instances $\mathcal{A}_1, \ldots, \mathcal{A}_t$ of the algorithm of Corollary 9.3.19. We define a sequence $(s_r)_{r=0}^t$ so that $\mathcal{A}_r$ works with a fixed delay $\Delta_{s_{r-1}}$, it is given $T[1..s_r) \cdot \perp^{\Delta_{s_{r-1}}}$, and it reports k-mismatch occurrences $i \in [s_{r-1}..s_r)$. Specifically, we set $s_0 = 1$ and $s_r = s_{r-1} + \Delta_{s_{r-1}}$, with t chosen as the smallest integer such that $s_t > d$. Note that $s_r - s_{r-1} = \Delta_{s_{r-1}} \geq \Delta_1$ implies $t \leq 1 + \frac{d}{\Delta_1} = O(1)$.

We assign three different roles to the algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_r$: *passive*, *active*, and *inactive*. While we process $T[j]$, the algorithm $\mathcal{A}_r$ is passive if $j < s_r$, active if $j \in [s_r..s_{r+1})$, and inactive if $j \geq s_{r+1}$. Our invariant is that, once we process $T[j]$, each passive algorithm $\mathcal{A}_r$ has already received $T[1..j]$, the unique active algorithm $\mathcal{A}_r$ has

already received $T[1..s_r)\cdot\perp^{1+i-s_{r-1}}$, where i is the largest integer such that $i+\Delta_i \le j$, and each inactive algorithm $\mathcal{A}_r$ has already received its entire input, that is, $T[1..s_r)\cdot\perp^{\Delta_{s_{r-1}}}$.

Upon receiving $T[j]$, we simply forward $T[j]$ to all passive algorithms. Moreover, if $j = i+\Delta_i$ for some $i \in [1..d]$, we feed the active algorithm with $\perp$ so that it checks whether i is a k-mismatch occurrence of P in T and, upon request, outputs $S_P^k[i]$.

Let us argue that this approach is correct from the perspective of a fixed algorithm $\mathcal{A}_r$. As we process $T[1..s_r)$, the algorithm is passive, and it is fed with subsequent characters of T. For $j = s_r - 1$, the position $i = s_{r-1} - 1$ is the maximum one such that $i + \Delta_i \le j$. Consequently, the input $T[1..s_r)$ already satisfies the invariant for passive algorithms. For subsequent iterations $j \in [s_r..s_{r+1})$, as $\mathcal{A}_r$ is active, it receives $\perp$ whenever i increases, so its input stays equal to $T[1..s_r)\cdot\perp^{1+i-s_{r-1}}$. The length of this string is $s_r + i - s_{r-1} = i + \Delta_{s_{r-1}}$, so the algorithm indeed checks whether i is a k-mismatch occurrence of P in T at each such iteration (recall that its fixed delay is $\Delta_{s_{r-1}}$), and it satisfies the invariant for active algorithms. Once we reach $j = s_{r+1}-1$, we have $i = s_r - 1 = s_{r-1} + \Delta_{s_{r-1}} - 1$, so the input becomes $T[1..s_r)\cdot\perp^{\Delta_{s_{r-1}}}$, and it already satisfies the invariant for inactive algorithms. The state of inactive algorithms does not change, so this invariant remains satisfied as $\mathcal{A}_r$ stays inactive indefinitely.

The time and space complexity analysis follows from the fact that $t = O(1)$. □

9.3.4.2 Algorithm

We now show how to use the data structure of Lemma 9.3.20 to implement our filtering procedure using low space. For each $j \in [1..\lfloor \log n \rfloor]$, let P_j denote the prefix of the text of length $\ell_j = 2^j$, i.e., $P_j = T[..2^j]$. We search for k-mismatch occurrences of P_j in $T_j = T(3\ell_j/2..4\ell_j]$. As argued below, this allows filtering positions in $(3\ell_j..6\ell_j]$. Additionally, our choice of $(\ell_j)_j$ ensures that we do not miss any k-mismatch square when running our search for every P_j in parallel.

▷ Claim 9.3.21. For each $j \in [1..\lfloor \log n \rfloor]$, let Occ_j be the set of k-mismatch occurrences of P_j in $T_j = T(3\ell_j/2..4\ell_j]$. If $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$ and $2i \in [3\ell_j..6\ell_j)$, then $p = i - \ell_j/2 \in \mathsf{Occ}_j$. Note that here, p is an index in T_j, and is therefore offset by $3\ell_j/2$ compared to indices in T.

Proof. Since $\ell_j \le i$, Observation 9.3.16 implies that $i+\ell_j$ is a k-mismatch occurrence of P_j in T. Moreover, when $2i \in [3\ell_j..6\ell_j)$, we have $3\ell_j/2 \le i \le 3\ell_j$; therefore, that k-mismatch occurrence of P_j is fully contained within T_j, and it ends at positions $i+\ell_j-3\ell_j/2 = i-\ell_j/2$ of T_j. □

In what follows, we use p to denote indices in T_j, whereas i denotes indices in the original text T. As $T_j = T(3\ell_j/2..4\ell_j]$, the correspondence is given by $i = p + 3\ell_j/2$. In other words, we only need to compute $\mathsf{hd}_{\le k}(T[..2i], \mathrm{SQ})$ when $i - \ell_j/2 \in \mathsf{Occ}_j$. As noted in Property 9.2.4, it suffices to know the sketches $\mathsf{sk}_k^{\mathsf{hd}}(T(i..2i])$ and $\mathsf{sk}_k^{\mathsf{hd}}(T[..i])$. We store $\mathsf{sk}_k^{\mathsf{hd}}(P_j) = \mathsf{sk}_k^{\mathsf{hd}}(T[..\ell_j])$ as well as $s_j = \mathsf{sk}_k^{\mathsf{hd}}(T[..3\ell_j/2])$ and maintain $\mathsf{sk}_k^{\mathsf{hd}}(T[..2i])$ in a rolling manner as we receive the characters of the text.

We use the algorithm of Lemma 9.3.20, asking for k-mismatch occurrences of P_j in T_j, to report $\mathsf{sk}_k^{\mathsf{hd}}(T_j[..i-\ell_j]) = \mathsf{sk}_k^{\mathsf{hd}}(T(\ell_j..i])$ for every $i \in \mathsf{Occ}_j$. The delay sequence is specified as $\Delta_p = p - \ell_j/2$ for $p \in [\ell_j..5\ell_j/2)$ so that the conditions of Lemma 9.3.20 are satisfied. (For $p < \ell_j$, we can assume $\Delta_p = \Delta_{\ell_j} = \ell_j/2$; anyway, there cannot be a k-mismatch occurrence of P_j before position ℓ_j.) This way, for every $i \in [3\ell_j/2..3\ell_j)$, we receive $S_{P_j}^k[i+\ell_j]$ (which corresponds to a potential k-mismatch occurrence starting at

position $i+1$) while processing $T_j[p+\Delta_p]$ for $p = i+\ell_j-3\ell_j/2 = i-\ell_j/2$. As $\Delta_p = p-\ell_j/2$, this corresponds to position $p' = 2p-\ell_j/2$ in T_j, or position $i' = 2p+\ell_j = 2i$ in T, i.e., this happens precisely as we are processing $T[2i]$. See Fig. 9.1 for an illustration of the above. If $S^k_{P_j}[i+\ell_j]$ is blank, we move on to the next position. Otherwise, we retrieve the sketch $\mathsf{sk}^{\mathsf{hd}}_k(T_j[..i]) = \mathsf{sk}^{\mathsf{hd}}_k(T(3\ell_j/2..i])$, combine it with $s_j = \mathsf{sk}^{\mathsf{hd}}_k(T[..3\ell_j/2])$ and $\mathsf{sk}^{\mathsf{hd}}_k(T[..2i])$ to obtain $\mathsf{sk}^{\mathsf{hd}}_k(T[..i])$ and $\mathsf{sk}^{\mathsf{hd}}_k(T(i..2i])$, and use the latter two sketches to compute $\mathsf{hd}_{\le k}(T[..i], T(i..2i])$, which is equal to $\mathsf{hd}_{\le k}(T[..2i], \mathrm{SQ})$ by Property 9.2.4.

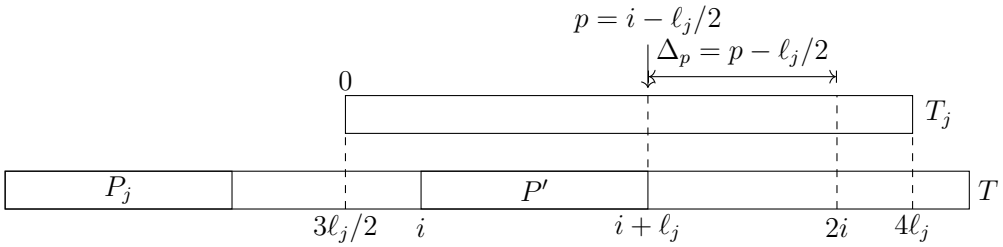

Figure 9.1: Illustration of our filtering procedure. Here, P' is a k-mismatch occurrence of P_j at position $i+\ell_j$ in T and position $p = i-\ell_j/2$ in T_j, reported with delay $\Delta_p = p-\ell_j/2$ in T_j, hence it arrives at time $2i$ in T.

We proceed with the complexity analysis of our algorithm. The k-mismatch pattern matching algorithm of Lemma 9.3.20 uses $O(k\log^2 n)$ bits of space and $\tilde{O}(k)$ time per character, and we maintain $O(\log n)$ instances of this algorithm. However, since all the patterns P_j are prefixes of T, the instances can share the pattern processing phase. Moreover, since any position is contained in at most three fragments $T[\ell_j..6\ell_j)$ (each such fragment follows P_j and contains T_j), at most three instances contribute to the time and space complexity at any given moment. Thus, the entire algorithm uses $O(k\log^2 n)$ bits of space and $\tilde{O}(k)$ time per character, which completes the proof of Theorem 9.3.15.

Our streaming algorithm for k-LED-SQ (Theorem 9.3.23) relies on the streaming algorithm for k-LHD-SQ. It requires testing $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$ only for selected positions i, and thus it benefits from the following variant of Theorem 9.3.15:

Proposition 9.3.22. *There is a randomized streaming algorithm that, given a string $T \in \Sigma^n$, upon receiving $T[2i]$, can be requested to test whether $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$ and, if so, report the mismatch information between $T[..2i]$ and a closest square. The algorithm uses $O(k\log^2 n)$ bits of space and processes each character in $\tilde{O}(\sqrt{k})$ or $\tilde{O}(k)$ time, depending on whether the request has been issued at that character.*

Proof. We follow the algorithm above with minor modifications. First, instead of maintaining $\mathsf{sk}^{\mathsf{hd}}_k(T[..2i])$ explicitly, we apply [107, Fact 4.4], which uses $O(k\log n)$ bits of space, takes $O(\log^2 n)$ time per character, and reports $\mathsf{sk}^{\mathsf{hd}}_k(T[..2i])$ on demand in $O(k\log^2 n)$ time.

To process a request concerning position $2i$, we retrieve $\mathsf{sk}^{\mathsf{hd}}_k(T[..2i])$ and ask the pattern-matching algorithm of Lemma 9.3.20 to output $S^k_{P_j}[i]$ (normally, the algorithm only reports whether i is a k-mismatch occurrence of P_j in T_j). In this case, we build $\mathsf{sk}^{\mathsf{hd}}_k(T[..i])$ and $\mathsf{sk}^{\mathsf{hd}}_k(T(i..2i])$ as in algorithm above. The decoding algorithm not only results in $\mathsf{hd}_{\le k}(T[..i], T(i..2i]) = \mathsf{hd}_{\le k}(T[..2i], \mathrm{SQ})$ but, if $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$, also the underlying mismatch information.

The space complexity of the modified algorithm is still $O(k\log^2 n)$ bits. The running time is $\tilde{O}(\sqrt{k})$ if we do not ask the algorithm to test $\mathsf{hd}(T[..2i], \mathrm{SQ}) \le k$ and $\tilde{O}(k)$ if we do. □

9.3.5 A streaming algorithm for k-LED-SQ

Theorem 9.3.23. *There is a randomized streaming algorithm that solves the k-LED-SQ problem for a string of length n using $\tilde{O}(k^2)$ bits of space and $\tilde{O}(k^2)$ time per character.*

Let $U = T[..i]$ be such that $\mathsf{ed}(U, \mathrm{SQ}) \le k$. By Property 9.2.10, there exist strings V, W such that $U = VW$ and $\mathsf{ed}(V, W) \le k$. Let $G = \mathbb{G}(V)$ and $G' = \mathbb{G}(W)$. Under the assumptions of Fact 9.3.3, we have $|G| = |G'| = s$ and G and G' differ at at most k indices, hence the Hamming distance between $\mathsf{enc}(G)\mathsf{enc}(G')$ and SQ is at most $k\mu = \tilde{O}(k^2)$. Therefore, our idea is to use the algorithm of Theorem 9.3.15 with parameter $K = k\mu$ to detect approximate squares in the stream of encoded grammars, and use this information to find positions i in the text where $\mathsf{ed}(T[..i], \mathrm{SQ}) \le k$. We first discuss the reduction to k-LHD-SQ, and then describe how to use it to solve k-LED-SQ.

Reduction to k-LHD-SQ. Two issues arise for the reduction to k-LHD-SQ: the grammar decomposition of U is not necessarily equal to the concatenation of G and G', and adding a character to the text may reduce the size of the decomposition, hence the conversion of the stream of characters to a stream of grammars is not trivial.

Let $G'' = \mathbb{G}(U)$ and $q = |G''|$. We assume w.l.o.g. that $q \ge 8\tau$ (which implies $s \ge 2\tau$); otherwise, we store all the grammars in G'' explicitly (taking $O(\tau\mu) = \tilde{O}(k)$ bits of space), and do not use the following reduction to k-LHD-SQ to find all the differing grammars, but use exhaustive pairwise comparison.

While G'' is not necessarily equal to GG', it follows from Corollary 9.3.5 that they share long prefixes and suffixes, namely $G''[1..s-\tau] = G[1..s-\tau]$, $G''(q-s+\tau..q] = G'(\tau..s]$ and $2s \le q + 2\tau$. In other words, G'' is equal to GG' with at most $4\tau = \tilde{O}(1)$ modified, inserted or deleted grammars. Furthermore, while appending a character to the text may reduce the number of grammars in its decomposition, right-committed grammars will not change when appending a character. Therefore, we can build a monotone stream of grammars by including only right-committed grammars, i.e., grammars whose index has been at τ positions away from the end of the decomposition at some point.

We decompose G'' as follows (see Fig. 9.2 for an illustration):

$$A = G''[1..\tau] \qquad D = G''(q-s+\tau..q-\tau]$$
$$B = G''(\tau..s-\tau] \qquad E = G''(q-\tau..q]$$
$$C = G''(s-\tau..q-s+\tau]$$

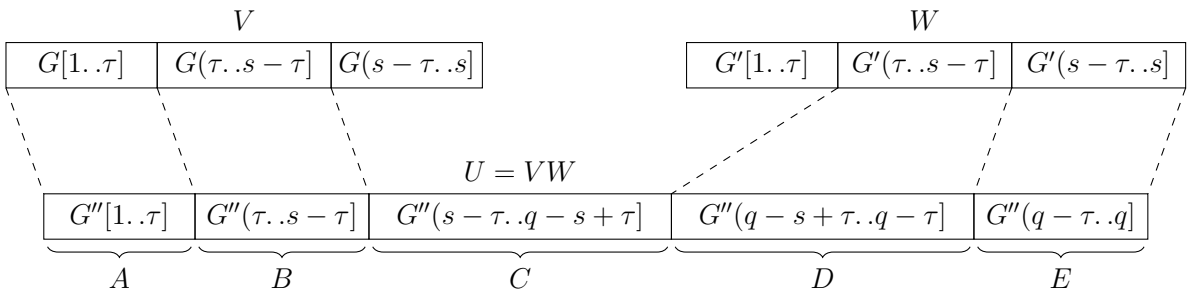

Figure 9.2: Decomposition of $U = VW$. Dashed lines indicate equal grammar sequences in the grammar decompositions of U, V or W.

By Corollary 9.3.5, we have the following identities: $A = G[1..\tau]$, $B = G(\tau..s-\tau]$, $D = G'(\tau..s-\tau]$, and $E = G'(s-\tau..s]$. Furthermore, by Fact 9.3.3, $\mathsf{hd}(B, D) \le k$,

where the Hamming distance between two equal-length sequences of grammars is defined as the number of positions where they differ. A and E are stored explicitly, which takes $O(\tau \cdot \mu) = \tilde{O}(k)$ space.

Further, for $c \in [1..4\tau]$, define $\hat{A}_c$ as follows:

$$\hat{A}_c = \begin{cases} A(\tau - c..], & \text{if } c \leq \tau, \\ \#^{c-\tau}A, & \text{otherwise,} \end{cases}$$

where $\#$ denotes the dummy (empty) grammar.

If G, G' are two grammar sequences, we define the Hamming distance between G and G', denoted $\mathsf{hd}(G, G')$, as the number of indices $i = 1, \ldots, s$, such that $G[i] \neq G'[i]$ if G and G' have the same length s, and $+\infty$ otherwise.

▷ Claim 9.3.24. If $\mathsf{ed}(U, \mathrm{SQ}) \leq k$, then there exists $c \leq 4\tau$ such that $\mathsf{hd}(\hat{A}_c B, CD) \leq k + 4\tau$.

Proof. For $c = |C| \leq 4\tau$, we have $|\hat{A}_c| = c$, and $\mathsf{hd}(\hat{A}_c B, CD) = \mathsf{hd}(\hat{A}_c, C) + \mathsf{hd}(B, D) \leq 4\tau + k$. □

For each $c \in [1..4\tau]$, we create a stream $T_c = \hat{A}_c BCD$ from a stream of $ABCD$ (by either dropping the first $\tau - c + 1$ values if $c \leq \tau$, or by inserting $c - \tau$ dummy grammars at the start otherwise) and apply the algorithm for k-LHD-SQ (Theorem 9.3.15) with parameter $K = (k + 4\tau)\mu = \tilde{O}(k^2)$ to it, which returns every position where $\mathsf{hd}(\hat{A}_c B, CD) \leq k + 4\tau$. We use this information as a filter before testing whether $\mathsf{ed}(U, \mathrm{SQ}) \leq k$.

Finally, notice that E contains τ grammars, which may be more than just the right-active grammars, as the size of the decomposition may shrink after appending a character. We only want grammars up to index $q - \tau$ to be in the stream of right-committed RLSLPs, i.e., we need to *remove* grammars from the stream when the size of the decomposition is reduced. To circumvent this issue, we simply store the grammars output by the algorithm for each of the τ latest updates. When the grammar decomposition shrinks, we simply go back to the corresponding grammars output and proceed from there. This takes $\tau \cdot \tilde{O}(k^2) = \tilde{O}(k^2)$ bits of space.

Computing the edit distance between V and W. Using Proposition 9.3.22 and Fact 9.3.7, we can also retrieve from the algorithm for k-LHD-SQ the list of mismatches between $\mathsf{enc}(\hat{A}_c BCD)$ and the closest square, from which we can extract the mismatching grammars in $\hat{A}_c B$ and CD. As we store A explicitly, we can assume to know every grammar of C.

We would like to compute $d = \mathsf{ed}_{\leq k}(V, W)$. Under the assumption of Fact 9.3.3, we have

$$\begin{aligned} \mathsf{ed}(V, W) &= \sum_{i=1}^{s} \mathsf{ed}(\mathsf{eval}(G[i]), \mathsf{eval}(G'[i])) \\ &= \mathsf{ed}(\mathsf{eval}(G[..\tau]), \mathsf{eval}(G'[..\tau])) \\ &\quad + \sum_{j : B[j] \neq D[j]} \mathsf{ed}(\mathsf{eval}(B[j]), \mathsf{eval}(D[j])) + \mathsf{ed}(\mathsf{eval}(G(s-\tau..]), \mathsf{eval}(G'(s-\tau..])). \end{aligned}$$

As for any strings X, X', Y, Y' we have $\mathsf{ed}(XY, X'Y') \leq \mathsf{ed}(X, X') + \mathsf{ed}(Y, Y')$, we can combine any two terms of the above sum by concatenating the corresponding grammars

in the eval function. Therefore, we have

$$\mathsf{ed}(V,W) = \mathsf{ed}(\mathsf{eval}(EA), \mathsf{eval}(G(s-\tau..s]G'[1..\tau])) + \sum_{j:B[j]\neq D[j]} \mathsf{ed}(\mathsf{eval}(B[j]), \mathsf{eval}(D[j])).$$

Finally, note that $\mathsf{eval}(G[s-\tau+1..s]G'[1..\tau]) = \mathsf{eval}(C)$ (see Fig. 9.2), and thus we have

$$\mathsf{ed}(V,W) = \mathsf{ed}(\mathsf{eval}(EA), \mathsf{eval}(C)) + \sum_{j:B[j]\neq D[j]} \mathsf{ed}(\mathsf{eval}(B[j]), \mathsf{eval}(D[j])).$$

Both $\mathsf{eval}(EA)$ and $\mathsf{eval}(C)$ can be represented by RLSLPs of size $\tilde{O}(k)$ obtained by concatenating the RLSLPs $G[s-\tau+1], \ldots, G[s], G'[1], \ldots, G'[\tau]$ and the RLSLPs in C respectively. We can now apply the algorithm of Ganesh et al. [156] to compute each of the terms in the sum above in $\tilde{O}(k^2)$ time and space.

Complexity analysis. Our algorithm runs $\tilde{O}(1)$ copies of the algorithm of Theorem 9.3.15 in parallel, with distance parameter $K = (k+4\tau)\mu = \tilde{O}(k^2)$: this uses $\tilde{O}(k^2)$ bits of space. Upon receiving the ith character of the text, we use the algorithm of Corollary 9.3.6 to update the decomposition of the text, which takes time $\tilde{O}(k)$. We add all but the last τ grammars to the stream of right-committed grammars. The algorithm of Corollary 9.3.6 may add up to $4\tau\lambda = \tilde{O}(1)$ grammars to the decomposition of the text, hence we commit at most $\tilde{O}(1)$ grammars per character of the text. Note that we only need to know whether the stream of right-committed grammars is close to a square after pushing all symbols in the encoding of the grammars, and do not need the intermediate results. Therefore, we can use the algorithm of Proposition 9.3.22 to push the at most $4\tau\lambda\mu = \tilde{O}(k)$ symbols in the encoding of the grammars into the k-LHD-SQ subroutines using $\tilde{O}(k\sqrt{K}+K) = \tilde{O}(k^2)$ time. After adding all the new right-committed grammars, if any copy of the k-LHD-SQ algorithm signals that the current stream of grammars is within distance K of SQ, we run the above algorithm to compute the distance between $T[..i]$ and SQ, which costs $\tilde{O}(k^2)$ time and space.

Finally, the algorithm explicitly stores the RLSLPs in A, the last τ grammars and the last τ outputs of each k-LHD-SQ subroutine, which uses $\tau\mu+\tau\mu+2\tau\mu+\tau\tilde{O}(k^2) = \tilde{O}(k^2)$ bits of space in total.

Overall, our algorithm uses $\tilde{O}(k^2)$ time per character and $\tilde{O}(k^2)$ bits of space.

9.4 Deterministic Read-Only Algorithms for the Hamming Distance Problems

In this section, we present deterministic read-only algorithms for k-LHD-PAL and k-LHD-SQ. We start by recalling structural results for k-mismatch occurrences used by the algorithms.

9.4.1 Structure of k-mismatch occurrences

In exact pattern matching, the structure of occurrences is related to periodicity [134]; in approximate pattern matching, the structure of *approximate* occurrences is related to *approximate* periodicity, which is defined as follows.

Definition 9.4.1 ([89]). *A string U is d-*mismatch periodic *if there exists a* primitive *string Q such that $|Q| \leq |U|/128d$ and $\mathsf{hd}(U, Q^\infty) \leq 2d$. Such a string Q is called the d-*mismatch period *of U.*

The condition $|Q| \leq |U|/128d$ implies that Q is equal to some substring of U; hence, given the starting and ending positions of Q in U and random access to U, we can simulate random access to Q. Furthermore, the approximate period of a string is related to the approximate period of its long prefixes.

Fact 9.4.2 (From [207, Claim 7.1]). *Let U and V be strings such that U is a prefix of V, and $|V| \leq 2|U|$. If U is d-mismatch periodic with d-mismatch period Q, then V either is not d-mismatch periodic or has the same d-mismatch period Q.*

Charalampopoulos et al. [89] showed that the set of k-mismatch occurrences has a very regular structure:

Fact 9.4.3 (See [89, Section 3]). *Let P and T be two strings such that $|P| \leq |T| \leq 3/2|P|$.*

1. *If P is not k-mismatch periodic, then there are $O(k)$ k-mismatch occurrences of P in T.*
2. *If P is k-mismatch periodic with period Q, then any two k-mismatch occurrences $i \leq i'$ of P in T satisfy $i \equiv i' \pmod{|Q|}$ and $\mathsf{hd}(T(i - |P|..i'], Q^\infty) \leq 3k$.*

They also presented efficient offline algorithms for computing the k-mismatch period and the k-mismatch occurrences in the so-called PILLAR model, which we briefly present here. In this model, one is given a family of strings $\mathcal{X}$ for preprocessing. The elementary objects are fragments $X[i..j]$ of strings $X \in \mathcal{X}$. Given elementary objects S, S_1, S_2, the PILLAR operations are:

1. $\mathsf{Access}(S, i)$: Assuming $i \in [1..|S|]$, retrieve $S[i]$.
2. $\mathsf{Length}(S)$: Retrieve the length $|S|$ of S.
3. $\mathsf{LCP}(S_1, S_2)$: Compute the length of the longest common prefix of S_1 and S_2.
4. $\mathsf{LCP}^R(S_1, S_2)$: Compute the length of the longest common suffix of S_1 and S_2.
5. $\mathsf{IPM}(S_1, S_2)$: Assuming that $|S_2| \leq 2|S_1|$, compute the set of the starting positions of occurrences of S_1 in S_2, which by Fine and Wilf periodicity lemma [134] can be represented as one arithmetic progression.

This model allows abstracting string algorithms in terms of the above operations, without considering the details of their implementation. The complexity of an algorithm is expressed in terms of the asymptotic number of PILLAR operations, plus some non-string operations, such as sorting numbers. Depending on the specific use case, we can choose an optimized implementation of these operations to minimize complexity, whether in the RAM model, on a quantum computer, in streaming, etc.

For example, Charalampopoulos et al. [89, Lemma 4.4] showed that the d-mismatch period of a string u can be computed using $O(d)$ time and space plus $O(d)$ PILLAR operations. In the read-only model, we can implement all PILLAR operations to run in linear time using $O(\log m)$ bits of space. The first four operations are implemented naively with linear scans; to implement the IPM operation, we use the algorithm of Rytter [262]. As a corollary, we immediately obtain:

Corollary 9.4.4. *Given random access to a string U, testing whether it is d-mismatch periodic, and, if so, computing its d-mismatch period, can be done using $O(d|U|)$ time and $O(d)$ space.*

9.4.2 Pattern matching with k mismatches in the read-only model

The above implementation of the PILLAR operations further implies an offline algorithm that finds all k-mismatch occurrences of P in T in $\tilde{O}(k^2 \cdot |T|)$ time and $\tilde{O}(k^2)$ space (see [89, Main Theorem 8]). Nevertheless, we provide a more efficient online algorithm that additionally provides the mismatch information for every k-mismatch occurrence of P.

Theorem 9.4.5. *There is a deterministic online read-only algorithm that finds all k-mismatch occurrences of a length-m pattern P within a text T using $O(k \log m)$ space and $O(k \log m)$ time per character. The algorithm outputs the mismatch information along with every reported k-mismatch occurrence of P.*

Consistently with the streaming algorithm of [107], our algorithm uses a family of exponentially-growing prefixes to filter out candidate positions. However, in order to use the structural properties of Fact 9.4.3 efficiently, we construct a different family $\mathcal{P}$ to ensure that we are either working in an approximately periodic region of the text or processing an aperiodic prefix.

We first add to $\mathcal{P}$ the prefixes $R_j = P[..\min\{m, \lfloor(3/2)^j\rfloor\}]$ for $j \in [0..\lceil\log_{3/2} m\rceil]$. If R_j is k-mismatch periodic but R_{j+1} is not, we also add to $\mathcal{P}$ the shortest extension of R_j that is not k-mismatch periodic. Hereafter, let $\mathcal{P} = (P_j)_{j=1}^{t}$ denote the resulting sequence of prefixes, sorted in order of increasing lengths, and let $\ell_j = |P_j|$ for every $j \in [1..t]$.

▷ Claim 9.4.6. The sequence $\mathcal{P} = (P_j)_{j=1}^{t}$ satisfies the following properties:
1. $P_1 = P[1]$ and $P_t = P$,
2. $t = |\mathcal{P}| = O(\log m)$,
3. for every $j \in [1..t)$, we have $\ell_{j+1} \leq 3\ell_j/2$,
4. for every $j \in [1..t)$, if P_j is k-mismatch periodic with period Q_j, then we have $\mathsf{hd}(P_{j+1}, Q_j^\infty) \leq 2k+1$.

Proof. Properties (1), (2), and (3) are straightforward. For (4), there are two possible cases: if P_{j+1} is k-mismatch periodic, Fact 9.4.2 implies that P_{j+1} has the same k-mismatch period Q_j as P_j, that is $\mathsf{hd}(P_{j+1}, Q_j^\infty) \leq 2k$. Otherwise, by construction, P_{j+1} is the shortest extension of P_j that is not k-mismatch periodic. By minimality, removing its last character yields a k-mismatch periodic prefix, and by Fact 9.4.2, it has the same k-mismatch period Q_j as P_j, i.e., we have $\mathsf{hd}(P[..\ell_{j+1}), Q_j^\infty) \leq 2k$. Adding one more character to $P[..\ell_{j+1})$ can increase the Hamming distance by at most one. □

Processing the pattern. We first preprocess the pattern: in this phase, we build the set $\mathcal{P}$ and, for each k-mismatch periodic prefix $P_j \in \mathcal{P} \setminus \{P\}$, we also compute its k-mismatch period Q_j (represented as a fragment of P_j) and the mismatch information $\mathsf{MI}(P_{j+1}, Q_j^\infty)$. Below, we describe the preprocessing procedure.

For each index $j \in [1..\lceil\log_{3/2} m\rceil]$, we add the prefix $R_j = P[..\min\{m, \lfloor(3/2)^j\rfloor\}]$ to $\mathcal{P}$. We apply Corollary 9.4.4 to test whether R_j is k-mismatch periodic and, if so, retrieve the period Q_j and compute $\mathsf{MI}(R_j, Q_j^\infty)$ using a linear scan. Then, we extend R_j to the right while maintaining the mismatch information with the appropriate prefix of Q_j^∞. We proceed until we reach length $|R_{j+1}|$ or $2k+1$ mismatches, whichever comes first. If we reach $2k+1$ mismatches before reaching length $|R_{j+1}|$, we add the obtained extension R'_j to $\mathcal{P}$, along with its mismatch information $\mathsf{MI}(R'_j, Q_j^\infty)$. Processing a given j takes $O(|R_{j+1}|k)$ time and $O(k)$ space, for a total of $O(mk)$ time and $O(k \log m)$ space across $j \in [1..\lceil\log_{3/2} m\rceil]$.

Processing the text. Our online algorithm processing the text T consists of $t = |\mathcal{P}|$ layers, where the j-th layer reports the k-mismatch occurrences of $P_j \in \mathcal{P}$, along with the corresponding mismatch information.

The first layer, responsible for $P_1 = P[1]$, is implemented naively in $O(1)$ space and time per character. Then, the layer $j+1$ receives the k-mismatch occurrences of P_j from layer j and outputs the k-mismatch occurrences of P_{j+1}. The filtering is based on the following simple observation:

Observation 9.4.7. *If there is a k-mismatch occurrence of P_{j+1} at position i in T, then there is a k-mismatch occurrence of P_j at position $i - \ell_{j+1} + \ell_j$ in T.*

In layer $j+1$, we partition T into blocks of length $b := \lceil \ell_j/2 \rceil$ and, for each block $T(rb..(r+1)b]$, $r = 0, \ldots, \lfloor m/b \rfloor$, use a separate subroutine to output k-mismatch occurrences of P_{j+1} at positions $i \in (rb..(r+1)b]$. This subroutine receives the k-mismatch occurrences of P_j at positions $i' = i - \ell_{j+1} + \ell_j$ in the interval $(rb - \ell_{j+1} + \ell_j..(r+1)b - \ell_{j+1} + \ell_j]$. We say that a subroutine is *active* while the algorithm reads $T(rb - \ell_{j+1} + \ell_j..(r+1)b]$; since $\ell_{j+1} \leq \frac{3}{2}\ell_j$, at most two subroutines are active at any given time. The implementation of the subroutine depends on whether P_j is k-mismatch periodic or not.

***P_j* is not *k*-mismatch periodic.** In this case, for every received k-mismatch occurrence i' of P_j, we receive the mismatch information $\mathsf{MI}(T(i' - \ell_j..i'], P_j)$ from the previous layer, and extend it letter by letter for $\ell_{j+1} - \ell_j$ positions, as long as there are less than k mismatches. If this is still the case for $i = i' + \ell_{j+1} - \ell_j$, we report a k-mismatch occurrence of P_{j+1} and output $\mathsf{MI}(T(i' - \ell_j..i], P[..\ell_j + i - i']) = \mathsf{MI}(T(i - \ell_{j+1}..i], P_{j+1})$. By Observation 9.4.7, no k-mismatch occurrence of P_{j+1} is missed. Moreover, Fact 9.4.3 guarantees that the subroutine receives in total at most $O(k)$ k-mismatch occurrences of P_j, hence this subroutine uses $O(k)$ space and $O(k)$ time per character.

***P_j* is *k*-mismatch periodic with period *Q_j*.** In this case, by Fact 9.4.3, the fragment of the text spanned by occurrences of P_j in the active block is similar (within $3k$ mismatches) to a prefix of Q_j^∞.

More precisely, we consider the leftmost k-mismatch occurrence $p \in (rb - \ell_{j+1} + \ell_j..(r+1)b - \ell_{j+1} + \ell_j]$ of P_j; we ignore all subsequent occurrences of P_j. We receive from layer j the mismatch information $\mathsf{MI}(T(p - \ell_j..p], P_j)$: we use the precomputed $\mathsf{MI}(P_j, Q_j^\infty)$ to construct $\mathsf{MI}(T(p - \ell_j..p], Q_j^\infty)$; by the triangle inequality, the size of this set is guaranteed to be at most $3k$. Then, as the algorithm receives subsequent characters of $T[i]$ for $i \in (p..(r+1)b]$, we maintain $\mathcal{M} = \mathsf{MI}(T(p - \ell_j..i], Q_j^\infty)$ as long as the number of mismatches does not exceed $6k+1$. Then, if $i \geq p + \ell_{j+1} - \ell_j$ and $i \equiv p + \ell_{j+1} - \ell_j \pmod{|Q_j|}$, we extract $\mathsf{MI}(T(i - \ell_{j+1}..i], Q_j^\infty)$ from $\mathcal{M}$ and use the precomputed mismatch information $\mathsf{MI}(P_{j+1}, Q_j^\infty)$ to compute $\mathsf{MI}(T(i - \ell_{j+1}..i], P_{j+1})$. If it is of size at most k, we report i as a k-mismatch occurrence of P_{j+1}.

Note that, if P_j is k-mismatch periodic, then we have precomputed $\mathsf{MI}(P_{j+1}, Q_j^\infty)$, because either P_{j+1} is k-mismatch periodic, and by Fact 9.4.2, P_{j+1} has the same period as P_j, or it is not k-mismatch periodic: in this case, it is a minimal aperiodic extension R'_j of P_j, and we explicitly computed its at most $2k+1$ mismatches to Q_j^∞.

As for the correctness, we argue that we miss no k-mismatch occurrence of P_{j+1} in T. Assume that there is such an occurrence at position $i \in (rb..(r+1)b]$. First, by Observation 9.4.7, $i - \ell_{j+1} + \ell_j$ is a k-mismatch occurrence of P_j in T. Furthermore, since $\mathsf{hd}(T(i - \ell_{j+1}..i], P_{j+1}) \leq k$ and $\mathsf{hd}(P_{j+1}, Q_j^\infty) \leq 2k+1$, we have $\mathsf{hd}(T(i - \ell_{j+1}..i], Q_j^\infty) \leq$

$3k+1$. Fact 9.4.3 further implies that $i-\ell_{j+1}+\ell_j \equiv p \pmod{|Q_j|}$ and $\mathsf{hd}(T(p-\ell_j\,..\,i-\ell_{j+1}], Q_j^\infty) \le 3k$. Consequently, by the triangle inequality, $\mathsf{hd}(T(p-\ell_j\,..\,i], Q_j^\infty) \le 6k+1$, and thus we compute $\mathsf{MI}(T(i-\ell_{j+1}\,..\,i], Q_j^\infty)$ and report i as a k-mismatch occurrence of P_{j+1}.

We conclude with the complexity analysis of this case: the working space is $O(k)$, dominated by the maintained mismatch information. Moreover, whenever we compute $\mathsf{MI}(T(i-\ell_{j+1}\,..\,i], P_j)$, the size of this set is, by the triangle inequality, at most $6k+1+2k+1 \le 8k+2$, and it can be computed in $O(k)$ time.

Complexity analysis. Overall, each subroutine of each level takes $O(k)$ space and $O(k)$ time per character. Since there are $t = O(\log m)$ levels and each level contains at most two active subroutines, the algorithm takes $O(k \log m)$ space and $O(k \log m)$ time per text character.

Furthermore, although our pattern preprocessing algorithm is an offline procedure, we can run it while the algorithm reads the first $m/2$ characters of the text. Then, while the algorithm reads further $m/2$ characters, it can process two characters at a time to catch up with the input stream. This does not result in any delay on the output because the leftmost k-mismatch occurrence of P is at position m or larger.

9.4.3 Read-only algorithm for k-LHD-PAL

In this section, we present an efficient online read-only algorithm for k-LHD-PAL.

Theorem 9.4.8. *There is a deterministic online algorithm that solves the* k-LHD-PAL *problem for a string of length* n *using* $O(k \log n)$ *space and* $O(k \log n)$ *worst-case time per character.*

The algorithm uses a filtering approach similar to that of the previous section. The filtering procedure selects a subset of positions where a prefix close to PAL can end, and we only need to verify these positions. We use the family $\mathcal{P} = \{P_j = T[\,..\lfloor (3/2)^j \rfloor] : j \in [1\,..\lfloor \log_{3/2} n \rfloor]\}$ of prefixes of the text, and let $\ell_j = |P_j|$, with the convention that $\ell_0 = 0$.

▷ Claim 9.4.9. Consider an index $j \in [1\,..\lfloor \log_{3/2} n \rfloor]$ and a position $i \in (2\ell_{j-1}\,..\,2\ell_j]$. If $\mathsf{hd}(T[\,..i], \mathrm{PAL}) \le k$, then i is a $2k$-mismatch occurrence of P_j^R in T. Moreover, $\mathsf{hd}(T[\,..i], \mathrm{PAL}) = \mathsf{hd}(T(i-i'\,..\,i], P_j[1\,..\,i')^R)$ for $i' = \lfloor i/2 \rfloor$.

Proof. Note that $i > 2\ell_{j-1} \ge \ell_j$ implies that P_j is a prefix of $T[\,..i]$ and, equivalently, P_j^R is a suffix of $T[\,..i]^R$. Property 9.2.3 implies $2 \cdot \mathsf{hd}(T[\,..i], \mathrm{PAL}) = \mathsf{hd}(T[\,..i], T[\,..i]^R) \ge \mathsf{hd}(T(i-\ell_j\,..\,i], P_j)$. Thus, if $\mathsf{hd}(T[\,..i], \mathrm{PAL}) \le k$, then i is a $2k$-mismatch occurrence of P_j in T. Since $T[\,..i']$ is a prefix of P_j, Property 9.2.3 further implies that

$$\mathsf{hd}(T[\,..i], \mathrm{PAL}) = \mathsf{hd}(T(i-i'\,..\,i], T[\,..i']^R) = \mathsf{hd}(T(i-i'\,..\,i], P_j[1\,..\,i')^R). \qquad \square$$

The algorithm constructs the family $\mathcal{P}$ as it reads the text. For each level j, we implement a subroutine responsible for identifying k-mismatch squares for the form $T[\,..i]$ for $i \in (2\ell_{j-1}\,..\,2\ell_j]$. This subroutine searches occurrences of P_j^R in $T(2\ell_{j-1}\,..\,2\ell_j]$ using the algorithm of Theorem 9.4.5.

We briefly discuss how to de-amortize the execution of the algorithm of Theorem 9.4.5. We run the pattern-preprocessing phase of this algorithm as well as its processing of $T[\,..2\ell_j)$ while reading $T[\ell_j\,..2\ell_{j-1})$. In this phase, we have random access to $T[\,..\ell_j] = P_j$, hence we can simulate random access to P_j^R. This first phase requires feeding the

algorithm $3\ell_j$ characters over $2\ell_{j-1} - \ell_j = \ell_{j-1}/2 = \ell_j/3$ positions, so we can feed the algorithm with $O(1)$ characters for every scanned character of T.

Then, while reading $T[2\ell_{j-1}..2\ell_j)$, we feed the pattern-matching algorithm with subsequent characters of T. For every reported $2k$-mismatch occurrence i of P_j^R in T_j, we retrieve the mismatch information $\mathsf{MI}(T(i-\ell_j..i], P_j^R)$ and obtain $\mathsf{MI}(T(i-i'..i], P_j[..i']^R)$ by removing the entries corresponding to the leftmost $\ell_j - i'$ positions. We report the size of this set (or $k+1$ if the size exceeds k) as $\mathsf{hd}_{\le k}(T[..i], \mathrm{PAL})$.

By Claim 9.4.9, all positions $i \in (2\ell_{j-1}..2\ell_j]$ such that $\mathsf{hd}(T[..i], \mathrm{PAL}) \le k$ pass the test and the distance $\mathsf{hd}(T[..i], \mathrm{PAL})$ is equal to the size of the set $\mathsf{MI}(T(i-i'..i], P_j[..i']^R)$. As for the complexity analysis, observe that, for each level j, we have $j = O(\log \ell_j)$, the pattern-matching algorithm uses $O(k \cdot j)$ space and takes $O(k \cdot j)$ time per character. Since, at any time, there is a constant number of active levels, the main algorithm uses $O(k \log n)$ space and takes $O(k \log n)$ time per character.

9.4.4 Read-only algorithm for k-LHD-SQ

Theorem 9.4.10. *There is a deterministic online algorithm that solves the k-LHD-SQ problem for a string $T \in \Sigma^n$ using $O(k \log n)$ space and $O(k \log n)$ worst-case time per character.*

Our algorithm is very similar to the pattern-matching algorithm of Theorem 9.4.5. We use the same sequence $\mathcal{P} = (P_j)_{j=1}^t$ of prefixes, now defined as prefixes of T. Again, we set $\ell_j = |P_j|$ for $j \in [1..t]$. Instead of Observation 9.4.7, we use Observation 9.3.16 to argue that our filtering procedure is correct.

Computing $\mathcal{P}$. We build $\mathcal{P}$ in an online fashion using the procedure described in Section 9.4.2, processing letters in batches of constant size so that so that the prefix P_j is constructed while scanning $T(\ell_j..\lceil 3\ell_j/2 \rceil]$. If P_j is k-mismatch periodic with period Q_j, then we also identify P_{j+1} and build $\mathsf{MI}(P_{j+1}, Q_j^\infty)$, which has size at most $2k+1$ by construction of $\mathcal{P}$. Across all indices $j \in [0..\lfloor \log_{3/2} n \rfloor]$, the preprocessing algorithm takes $O(k \log n)$ space and time per character (since no two indices are processed simultaneously).

Computing the distances. For each level $j \in [1..t]$, we implement a subroutine responsible for testing whether $T[..i]$ is a k-mismatch square for even positions $i \in [2\ell_j..2\ell_{j+1})$; this procedure is active as we read $T[\ell_j..2\ell_{j+1})$. As described above, the pattern P_j is identified while the algorithm reads $T(\ell_j..\lceil 3\ell_j/2 \rceil]$ and, if P_j is k-mismatch periodic, the period Q_j and the mismatch information $\mathsf{MI}(P_{j+1}, Q_j^\infty)$ are also computed at that time.

While reading $T[\lceil 3\ell_j/2 \rceil..2\ell_j)$, we launch the pattern-matching algorithm of Theorem 9.4.5 to report the k-mismatch occurrences of P_j in $T_j = T[\ell_j..\ell_j + \ell_{j+1})$. As no such occurrence appears before position $2\ell_j$ in T, we can use the same de-amortization technique to feed P_j and $T[\ell_j..2\ell_j)$ to the algorithm while reading $T[\ell_j..2\ell_j)$, feeding $O(1)$ character per character of the text. Then, while reading $T[2\ell_j..\ell_j + \ell_{j+1})$, we resume to a normal execution of the pattern matching algorithm, feeding it one character per position of T.

If the algorithm reports that $i' \in [2\ell_j..\ell_j + \ell_{j+1})$ is a k-mismatch occurrence of P_j in T with mismatch information $\mathsf{MI}(P_j, T(i'-\ell_j..i'])$, we use this output to decide whether $T[..2i]$ is a k-mismatch square, where $i = i' - \ell_j$. How we utilize this output depends

on whether P_j is k-mismatch periodic or not: if P_j is not k-mismatch periodic, then T_j contains $O(k)$ k-mismatch occurrences of P_j and storing them explicitly requires little space. When P_j is k-mismatch periodic, T_j must exhibit similar periodicity, which we can use to avoid storing all occurrences explicitly.

P_j is not k-mismatch periodic. In this case, for every received k-mismatch occurrence i' of P_j, define $i = i' - \ell_j$: we store the cardinality $\mathcal{N}_{i,i'}$ of the mismatch information $\mathcal{M}_{i,i'} = \mathsf{MI}(P_j, T(i..i'])$. Note that, by definition of P_j, $\mathcal{M}_{i,i'} = \mathsf{MI}(T[..\ell_j], T(i..i'])$. Then, as the algorithm receives the subsequent characters $T[p]$ for $p \in (i'..2i]$, we compare $T[p]$ to $T[p-i]$ to compute $\mathcal{N}_{i,p}$, the cardinality of $\mathcal{M}_{i,p} = \mathsf{MI}(T[..p-i], T(i..p])$, from $\mathcal{N}_{i,p-1}$. We have:

$$\mathcal{N}_{i,p} = \begin{cases} \mathcal{N}_{i,p-1} + 1 & \text{if } T[p] \neq T[p-i] \\ \mathcal{N}_{i,p-1} & \text{otherwise.} \end{cases}$$

When $p = 2i$, we have $\mathcal{M}_{i,p} = \mathsf{MI}(T[..i], T(i..2i])$: by Property 9.2.4, $\mathcal{N}_{i,2i}$ is equal to the distance between $T[..2i]$ and SQ, and we can report whether $T[..2i]$ is a k-mismatch square.

By Observation 9.3.16, no k-mismatch square $T[..2i]$ is missed. Moreover, Fact 9.4.3 guarantees that there are $O(k)$ k-mismatch occurrences of P_j, and thus we use $O(k)$ space and $O(k)$ time per character to process all of them.

P_j is k-mismatch periodic with period Q_j. In this case, we wait for the leftmost k-mismatch occurrence $i' \in [2\ell_j..\ell_j + \ell_{j+1})$ of P_j in T and ignore all the subsequent occurrences of P_j. Let $i = i' - \ell_j$. We use the received mismatch information $\mathsf{MI}(T(i..i'], P_j)$ and the preprocessed mismatch information $\mathsf{MI}(P_j, Q_j^\infty)$ to construct $\mathsf{MI}(T(i..i'], Q_j^\infty)$; by the triangle inequality, the size of this set is guaranteed to be at most $3k$.

As the algorithm receives subsequent characters of $T[p]$ for $p \in (i'..2\ell_{j+1})$, we maintain $\mathsf{MI}(T(i..p], Q_j^\infty)$ as long as the number of mismatches does not exceed $6k+1$. Then, for any even $p = 2p' \geq 2i$ such that $p' \equiv i \pmod{|Q_j|}$, we extract $\mathcal{M} = \mathsf{MI}(T(p'..p], Q_j^\infty)$ from $\mathsf{MI}(T(i..p], Q_j^\infty)$, and extract $\mathcal{M}' = \mathsf{MI}(T[..p'], Q_j^\infty)$ from the precomputed mismatch information $\mathsf{MI}(P_{j+1}, Q_j^\infty)$. We combine $\mathcal{M}$ and $\mathcal{M}'$ to compute $\mathsf{hd}(T[..p'], T(p'..p])$: if it is less than k, we report $T[..p]$ as a k-mismatch square.

For the correctness, by Observation 9.3.16, if $T[..p]$ is a k-mismatch square, then $p' + \ell_j$ is a k-mismatch occurrence of P_j. Fact 9.4.3 further implies that $p' + \ell_j \equiv i' \pmod{|Q_j|}$ and $\mathsf{hd}(T(i..p], Q_j^\infty) \leq 3k$. Consequently, $\mathsf{hd}(T(i..p], Q_j^\infty) \leq 6k+1$, and the position p is correctly reported.

We conclude with the complexity analysis of this procedure: the working space is $O(k)$, dominated by the maintained mismatch information which have size $O(k)$. It follows that the distances can be computed in $O(k)$ time.

Complexity analysis. Overall, each level j takes $O(k \log n)$ space and $O(k \log n)$ time per character, dominated by the pattern-matching algorithm of Theorem 9.4.5. However, since a constant number of levels are processed in parallel at any given time, the entire algorithm still uses $O(k \log n)$ space and $O(k \log n)$ time per character overall.

9.5 Deterministic Read-Only Algorithms for the Edit Distance Problems

In this section, we present read-only algorithms for k-LED-PAL and k-LED-SQ, and the edit-distance-related tools required to analyse them: an online read-only algorithm for k-error pattern matching occurrences and the structural results of k-error occurrences.

9.5.1 Structure of k-error occurrences

Approximate periodicity.

Definition 9.5.1 ([89]). *A string U is d-*error periodic *if there exists a* primitive *string Q such that $|Q| \leq |U|/128d$ and $\mathsf{ed}(U, Q^\infty) \leq 2d$. Such a string Q is called the d-*error period *of U.*[4]

Similarly to the Hamming distance, the condition $|Q| \leq |U|/128d$ implies that if U is d-error periodic with d-error period Q, then Q is equal to some substring of U. Furthermore, we exploit the following properties:

Fact 9.5.2 ([207]). *Suppose that a string X is a prefix of a string Y, where $|X| < |Y| \leq 2|X|$. If X is k-error periodic with k-error period Q, then either Y is not k-error periodic, or Y is k-error periodic with k-error period Q.*

Lemma 9.5.3. *Given random access to a string U, testing whether it is d-error periodic, and in the relevant case computing its d-error period, can be done using $O(|U|d^2)$ time and $O(d)$ space.*

Proof. Consider a partition of U into $4d$ substrings $S_1, \ldots, S_{4d}$ of equal length $|U|/4d$. At most $2d$ of them can contain an edit operation, hence the others must be equal to a substring of Q^∞, and as $|Q| \leq |U|/128d = |S_i|/32$, these S_i are (exactly) periodic.

This observation leads to the following algorithm: For $i \in [1..4d]$, we test whether S_i is (exactly) periodic: if it is, we compute its period Q'_i; otherwise, we go to the next i. Assume that we are in the periodic case, and let p_i denote the ending position of S_i in U. We construct the string S'_i by extending S_i to the left with copies of Q'_i until its length exceeds p_i, and test whether there exists an index j such that the edit distance between $U[..p_i]$ and $S'_i[j..]$ is at most $2d$. Let Q_i be the cyclic left rotation of Q'_i by $j \mod |Q'_i|$ positions: it is a candidate for the d-error period of U. We then test whether the edit distance between U and $(Q'_i)^\infty$ is at most $2d$: in the affirmative case, Q_i is a d-error period of U; otherwise, we go to the next i. If U is d-error periodic, at least one of the first $2d+1$ tests must return the d-error period.

Computing the exact period of a string R can be done in linear time using $O(\log n)$ bits of space, using the algorithm of Rytter [262] to find the first occurrence of R in $R[2..]R$. Computing the index that minimizes the edit distance upper bounded by d can be done in $O(|U|d)$ time and $O(d)$ space using the classical dynamic programming algorithm for the edit distance [279]. We process $O(d)$ substrings S_i sequentially, hence our algorithm takes $O(|U|d^2)$ time and $O(d)$ space. □

[4] The original definition of [89] considered the distance between U and a substring of Q^∞. Changing it does not break the structure of k-error occurrences, but is important for the read-only algorithms for k-LED-PAL and k-LED-SQ.

Structure of approximate occurrences.

Definition 9.5.4 (Chain of k-error occurrences)**.** *A sequence $\mathcal{C} = p_1 < \cdots < p_\ell$ of k-error occurrences of P in T forms a chain if the following two conditions are satisfied:*

1. *There exists an integer q such that $p_1, \ldots, p_\ell$ is an arithmetic progression with difference q;*
2. *There exists an integer $k' \leq k$ such that for every p_i, $\min_{j \leq p_i} \mathsf{ed}(P, T[j\,..p_i]) = k'$.*

The following fact follows from [89, Theorem 5.1, Claim 5.16, Claim 5.17], see also [207, Corollary III.5]:

Fact 9.5.5. *Let P, T be two strings such that $|T| \leq 3/2|P|$ and T starts and ends with a k-error occurrence of P. Then one of the following holds:*

1. *Either there are $O(k^2)$ k-error occurrences of P in T, or*
2. *There is a primitive string Q such that P is $2k$-error periodic with $2k$-error period Q, $\mathsf{ed}(T, Q^\infty) \leq 6k$, and the occurrences of P in T can be decomposed into $O(k^3)$ chains. For each chain, its difference equals $|Q|$ and the first position in it is within distance $10k$ from a multiple of $|Q|$.*

9.5.2 Online deterministic read-only algorithm for finding k-error occurrences

Fact 9.5.6 ([226])**.** *Given two strings U, V, there is a data structure that can be built using $O(k^2)$ LCP queries, occupies $O(k^2)$ space and allows retrieving, for any two prefixes U', V' of U, V respectively, the value of $\mathsf{ed}_{\leq k}(U', V')$ in $O(k)$ time.*

To be able to utilise this fact, we show how to answer LCP queries in k-error periodic strings:

Lemma 9.5.7. *Consider three strings $U, V, Q \in \Sigma^*$. Assume that Q is a primitive string and that there exist $O(k)$-length edit sequences $\mathsf{ES}_U, \mathsf{ES}_V$ between U and $Q^\infty[1\,..|U|]$ and V and $Q^\infty[1\,..|V|]$, respectively. There is a read-only algorithm that receives as an input U, V and $\mathsf{ES}_U, \mathsf{ES}_V$ and computes $\mathsf{LCP}(U[i\,..], V[j\,..])$ in time $O(k|Q|)$ and space $\tilde{O}(1)$.*

Proof. We can assume w.l.o.g. that $U[i] = V[j]$, otherwise the LCP is 0.

We consider two cases: (1) Both $U[i]$ and $V[j]$ are unedited by the edit sequences; (2) Either $U[i]$ is edited by ES_U or $V[j]$ is edited by ES_V.

In the first case, let i' (resp., j') be the length of $U[1\,..i]$ (resp., $V[1\,..j]$) after we apply ES_U (resp., ES_V) to it. These indices correspond to the position that $U[i]$ and $V[j]$ have in Q^∞. We perform an LCP query on $Q^\infty[i'\,..]$ and $Q^\infty[j'\,..]$. If $i' = j' \pmod{|Q|}$, the answer is $+\infty$, and otherwise, we compute the answer in a naive way, comparing $Q^\infty[i'\,..]$ and $Q^\infty[j'\,..]$ character-by-character. As Q is primitive, we will find a mismatch after performing $O(|Q|)$ comparisons in the latter case. Let $\ell = \mathsf{LCP}(Q^\infty[i'\,..], Q^\infty[j'\,..])$. If the prefixes of length ℓ of $U[i\,..]$ and $V[j\,..]$ are unedited by ES_U and ES_V, then $\mathsf{LCP}(U[i\,..], V[j\,..]) = \min\{\ell, |U[i\,..]|, |V[j\,..]|\}$. Otherwise, let t_U and t_V be indices such that $i + t_U$ (resp. $j + t_V$) is the leftmost edited character in $U[i\,..]$ (resp. in $V[j\,..]$), and let $t = \min\{t_U, t_V\}$: the LCP is equal to $t - 1 + \mathsf{LCP}(U[i + t\,..], V[j + t\,..])$, which we compute recursively.

Consider now the second case, when one of $U[i]$ or $V[j]$ is affected by an edit operation. As $U[i] = V[j]$, the LCP is equal to $1 + \mathsf{LCP}(U[i + 1\,..], V[j + 1\,..])$, which we can compute

with a recursive call. As the edit sequences have length $O(k)$, this case can occur at most $O(k)$ times.

An LCP query in Q^∞ (the first case) takes $O(k)$ time and is either the last step of the algorithm or is followed by a call to the second case of the algorithm. As the second case happens $O(k)$ times, the algorithm takes $O(k|Q|)$ time. □

Lemma 9.5.8. *There is a deterministic online read-only algorithm that finds all k-error occurrences of a length-m pattern P in a text T using $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortized time per character.*

Proof. We prove that, for every $d \in \mathbb{Z}_{\geq 0}$, there is an algorithm reporting the k-error occurrences with a delay exactly d.

If $d \geq \frac{m}{4}$, then we partition the text into disjoint blocks of $b = \lfloor \frac{m}{4} \rfloor$ characters. Consider a block $T(r-b..r]$. Having processed $T[r+d-b]$, we use the offline algorithm [89, Main Theorem 9] to retrieve the k-error occurrences of P ending within the block $T(r-b..r]$, reported as $O(k^3)$ chains. This costs $\tilde{O}(bk^4)$ time and uses $\tilde{O}(k^4)$ space. For every $i \in (r-b..r]$, while processing $T[i+d]$, we check if any of the chains contain an occurrence ending at position i and, if so, report the underlying distance $\min_j \mathsf{ed}_{\leq k}(P, T(j..i])$.

Now, suppose that $k \leq d < \frac{m}{4}$. We partition P into a prefix $L = P[1..m-4d]$ and a suffix $R = P(m-4d..m]$. We recursively report the occurrences of L with a delay of $5d-k \geq 4d$ and store them in a buffer of size $2k+1$. In particular, while the algorithm processes $T[i+d]$, we have access to $\min_j \mathsf{ed}_{\leq k}(L, T(j..i'])$ for all $i' \in [i-4d-k..i-4d+k]$. We also partition the text into disjoint blocks of d characters. Consider a block $T(r-d..r]$. Having received the entire block, we use the offline algorithm [89, Main Theorem 9] to retrieve the k-error occurrences of R ending within the block $T(r-d..r]$, reported as $O(k^3)$ chains. This costs $\tilde{O}(dk^4)$ time and uses $\tilde{O}(k^4)$ space. If R is far from periodic, it has $O(k^2)$ occurrences. Otherwise, R has a $2k$-error period Q and, if p and p' are the leftmost and rightmost positions in $T(r-d..r]$ where the reported occurrences end, then $T(p-4d-k..p']$ is at edit distance $O(k)$ from a substring of Q^∞. In that case, we retrieve the underlying edits as well as the $O(k)$ edits between R and a substring of Q^∞. In either case, while processing $T[i+d]$, if a k-edit occurrence of R ends at position i, we compute $\mathsf{ed}_{\leq k}(R, T(i'..i])$ for all $i' \in [i-4d-k..i-4d+k]$ using Fact 9.5.6. In the non-periodic case, we use the naive $O(d)$-time implementation of LCP queries. This costs $O(dk^2)$ time per occurrence and $O(k^4)$ amortized time per position. Otherwise, we use the $O(|Q| \cdot k)$-time implementation of LCP queries; see Lemma 9.5.7. This costs $O(|Q| \cdot k^3)$ time per occurrence and, since there are $O(k)$ occurrences per every $|Q|$ positions, $O(k^4)$ amortized time per position as well. We can combine $\mathsf{ed}_{\leq k}(R, T(i'..i])$ with $\min_j \mathsf{ed}_{\leq k}(L, T(j..i'])$ (minimizing over $i' \in [i-4d-k..i-4d+k]$) to obtain $\min_j \mathsf{ed}_{\leq k}(P, T(j..i])$.

Finally, if $d < k$, we partition P into a prefix $L = P[1..m-4k]$ and a suffix $R = P(m-4k..m]$. We recursively report the occurrences of L with a delay of $d+3k > 4d$ and store them in a buffer of size $2k+1$. In particular, while the algorithm processes $T[i+d]$, we have access to $\min_j \mathsf{ed}_{\leq k}(L, T(j..i'])$ for all $i' \in [i-5k..i-3k]$. While processing $T[i+d]$, we compute $\mathsf{ed}_{\leq k}(R, T(i'..i])$ for all $i' \in [i-5k..i-3k]$ using Fact 9.5.6 and the naive $O(k)$-time implementation of LCP queries. This costs $O(k^3)$ time per position. We can combine $\mathsf{ed}_{\leq k}(R, T(i'..i])$ with $\min_j \mathsf{ed}_{\leq k}(L, T(j..i'])$ (minimizing over $i' \in [i-4d-k..i-4d+k]$) to obtain $\min_j \mathsf{ed}_{\leq k}(P, T(j..i])$.

At each level of the recursive algorithm, we use $\tilde{O}(k^4)$ space and amortized time per character. The recursive calls decrease the pattern length and increase the delay by a factor of at least 4, so the depth of the recursion is $O(\log m)$. □

9.5.3 Read-only algorithm for k-LED-PAL

Theorem 9.5.9. *There is a deterministic online read-only algorithm that solves the k-LED-PAL problem for a string of length n using $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ time per character.*

Consider a family $\mathcal{P} = \{P_j : P_j = T[..2^j], j \in [1..\lfloor \log n \rfloor]\}$ of prefixes of the text. For each j, define $\ell_j = |P_j|$, $T_j = T[..\ell_{j+1})$, and Occ_j to be the set of $2k$-error occurrences of P_j^R in T_j at position greater than or equal to ℓ_j. Using Lemma 9.5.3, we can decide whether P_j is $2k$-error periodic and if it is, compute its $2k$-error period Q_j in $O(|P_j|k^2)$ time and $O(k)$ space. Using Ukkonen's algorithm [279], we can also compute a $O(k)$-length edit sequence ES_j from P_j to $(Q_j)^\infty$ in $O(k)$ amortized time per character and $\tilde{O}(k^2)$ space.

▷ Claim 9.5.10. If $\mathsf{ed}(T[..i], \mathrm{PAL}) \le k$, then $i \in \mathsf{Occ}_{j^*}$ for $j^* = \max\{j : \ell_j \le i\}$.

Proof. By Corollary 9.2.9, $\mathsf{ed}(T[..i], T[..i]^R) \le 2k$. We also have $\ell_j \le i \le 2\ell_j - 1$. Hence, $i \in \mathsf{Occ}_j$. □

Define $i' = \lfloor i/2 \rfloor$. By Corollary 9.2.9, to decide the edit distance between $T[..i]$ and PAL, it suffices to compute

$$e = \min_{j \in [i'-k..i'+k]} \{\min\{\mathsf{ed}_{\le k}(T[..j], T[j+1..]^R), \mathsf{ed}_{\le k}(T[..j], T[j+2..]^R)\}\}.$$

We will only compute this value if the index i is in the set Occ'_{j^*}, a superset of Occ_{j^*} that we define below, and otherwise we will output $k+1$.

For each j, we run the online algorithm of Lemma 9.5.8 that uses $\tilde{O}(k^4)$ amortized time per character and $\tilde{O}(k^4)$ space to find the set Occ_j of k-error occurrences of P_j^R in T_j. We start it immediately after receiving $T[\ell_j]$ and catch up by processing the first ℓ_j characters of T_j at once to be online. If P_j is not $2k$-error periodic, then by Fact 9.5.5, $|\mathsf{Occ}_j| = O(k^2)$ and we define $\mathsf{Occ}'_j = \mathsf{Occ}_j$. If P_j is $2k$-error periodic, let p_j be the leftmost position in Occ_j, and r_j the largest integer such that $\mathsf{ed}(T[p_j..p_j + r_j \cdot |Q_j|], (Q_j^R)^\infty) \le 12k$. Define $\mathsf{Occ}'_j = \{p_j + m \cdot |Q_j| + \Delta : 0 \le m \le r_j, |\Delta| \le 10k\}$. By Fact 9.5.5, $\mathsf{Occ}'_j \supseteq \mathsf{Occ}_j$. The integer r_j can be computed by running an instance of Ukkonen's online algorithm [279] for $T[p_j + 1..]$ and $(Q_j^R)^\infty$, which takes $O(k)$ amortized time per character and $\tilde{O}(k^2)$ space. If $i \le p_j + r_j \cdot |Q_j|$ is the current position, the algorithm also allows extracting an $O(k)$-length edit sequence ES'_j between $T[p_j..i]$ and $(Q_j^R)^\infty$ in $O(k)$ time.

Computing the distances. Let i be the current position and $j^* = \max\{j : \ell_j \le i\}$. Assume that $i \in \mathsf{Occ}'_j$. If P_j is not $2k$-error periodic, we compute e using k instances of Ukkonen's online algorithm [279], which takes $O(k^2)$ amortized time per character in total and $\tilde{O}(k^2)$ space. Otherwise, the value e is computed as follows. First, if $i = p_j$, we run another instance of Ukkonen's online algorithm [279] to compute an $O(k)$-length edit sequence ES''_j between $T(p_j - \ell_j - k..p_j]$ and $(Q_j^R)^\infty$, which must exist as p_j is a $2k$-error occurrence of P_j^R and P_j is k-error periodic with period Q_j. This takes $O(k)$ amortized time and $\tilde{O}(k^2)$ space. Now, to compute the value e, we extract $O(k)$-length edit sequences between $T[1..\ell_j + k]$ and Q_j^∞ and between $T(i - \ell_j - k..i]$ and $(Q_j^R)^\infty$ in $O(k)$ time from ES_j, ES'_j, and ES''_j, and then apply Fact 9.5.6 and Lemma 9.5.7 to compute e in $O(k^3|Q_j|)$ time and $\tilde{O}(k^2)$ space.

Summary. The algorithm of Lemma 9.5.8 uses $\tilde{O}(k^4)$ amortized time per character and $\tilde{O}(k^4)$ space. Processing aperiodic prefixes costs $\tilde{O}(k^2)$ amortized time per character and $\tilde{O}(k^2)$ space. To upper bound the complexity of processing periodic prefixes, note that we test $O(k)$ positions out of $|Q_j|$, and therefore use $O(k^4)$ amortized time per character and $\tilde{O}(k^2)$ space.

9.5.4 Read-only algorithm for k-LED-SQ

Theorem 9.5.11. *There is a deterministic online read-only algorithm that solves the k-LED-SQ problem for a string of length n using $\tilde{O}(k^4)$ bits of space and $\tilde{O}(k^4)$ amortized time per character.*

We first define a filtering family of prefixes $\mathcal{P}$ of the text. Start by considering the prefixes $R_j = T[..\lfloor(3/2)^j\rfloor]$, $j \in [1..\lfloor\log_{3/2} n\rfloor]$. If R_j is k-error periodic but R_{j+1} is not, add to $\mathcal{P}$ the shortest extension of R_j that is not k-error periodic. Hereafter, let $\mathcal{P} = \{P_j\}$ denote the resulting family of prefixes, sorted in order of increasing lengths. For each j, let $\ell_j = |P_j|$, $T_j = T[1..\ell_{j+1}+\ell_j]$, and Occ_j the set of $3k$-error occurrences of P_j in T_j. We build $\mathcal{P}$ as we read T. When $T[\ell_j]$ arrives, we add R_j to P_j and launch an instance of the patter-matching algorithm for T_j to compute Occ_j (we process the first ℓ_j characters of T_j at once to be online). Finally, we apply Lemma 9.5.3 to test R_j for $3k$-error periodicity, which requires $O(k^2)$ amortized time and $\tilde{O}(k)$ space. If R_j is $3k$-periodic, the lemma also allows to compute the period Q_j of R_j. We finish by computing the longest $3k$-error periodic extension R'_j of R_j: by Fact 9.5.2, the $3k$-error period of R'_j equals Q_j and therefore we can compute R'_j by running Ukkonen's algorithm [279] on T and Q_j, which requires $O(k)$ amortized time and $\tilde{O}(k^2)$ space. The algorithm also outputs $O(k)$-length edit sequence ES_j between R'_j and $Q_j^\infty[1..|R'_j|]$.

▷ Claim 9.5.12. If $\mathsf{ed}(T[..i], \mathrm{SQ}) \le k$, then $i' + \ell_{j^*} \in \mathsf{Occ}_{j^*}$, where $i' = \lfloor i/2\rfloor$ and $j^* = \max\{j : \ell_j + \lceil k/2\rceil \le i'\}$. Furthermore, if P_{j^*} is $3k$-periodic, then $\mathsf{ed}(T(i'+\ell_{j^*}..i], Q_{j^*}^\infty) \le 10k+1$.

Proof. By Corollary 9.2.11, we have $\mathsf{ed}(T[..i'+t], T[i'+t+1..i]) \le k$ for some t, $|t| \le \lceil k/2\rceil$. As P_{j^*} is a prefix of $T[..i'+t]$, we have $\mathsf{ed}(P_{j^*}, T[i'+t+1..i'+t+\ell_{j^*}+\Delta]) \le k$ for some $-k \le \Delta \le k$ and therefore by the triangle inequality $\mathsf{ed}(P_{j^*}, T[i'+t+1..i'+\ell_{j^*}]) \le 3k$. It follows that $\ell_{j^*} < i'+\ell_{j^*} \le \ell_{j^*+1}+\ell_{j^*}$ is a $3k$-error occurrence of P_{j^*} and hence $i'+\ell_{j^*} \in \mathsf{Occ}_{j^*}$. To show the second part of the claim, note that $i'+t \le \ell_{j^*+1}+k$. Additionally, by the construction of $\mathcal{P}$ and Fact 9.5.2 we have $\mathsf{ed}(P_{j^*}, (Q_{j^*})^\infty) \le 7k+1$. Applying the triangle inequality one more time, we obtain that $\mathsf{ed}(T[i'+t+1..i], (Q_{j^*})^\infty) \le 10k+1$, as removing equal-length suffixes of both strings cannot increase the edit distance. □

By Corollary 9.2.11, to get the edit distance between $T[..i]$ and SQ, it suffices to compute the value $e = \min_{j\in[i'-k..i'+k]} \mathsf{ed}_{\le k}(T[..j], T(j..])$. We will only compute this value if $i'+\ell_{j^*} \in \mathsf{Occ}'_{j^*} \supseteq \mathsf{Occ}_{j^*}$ to be defined below, and otherwise we output $k+1$. The set Occ'_{j^*} is defined differently depending on whether P_j is $3k$-error periodic.

If P_j is not $3k$-error periodic, $\mathsf{Occ}'_{j^*} = \mathsf{Occ}_{j^*}$ and by Fact 9.5.5, occupies $\tilde{O}(k^2)$ space. Using Lemma 9.5.8, Occ'_{j^*} can be computed in $\tilde{O}(k^4)$ amortized time per character and $\tilde{O}(k^4)$ space. If P_j is $3k$-error periodic with a period $|Q_j|$, Occ'_{j^*} is defined to be $\{p_{j^*} + t\cdot|Q_{j^*}| + \Delta : 0 \le t \le r_{j^*}, |\Delta| \le 10k\}$, where p_{j^*} is the leftmost position in Occ_{j^*} and r_{j^*} is the largest integer such that $\mathsf{ed}(T[p_{j^*}..p_{j^*}+r_{j^*}\cdot|Q_{j^*}|], (Q_{j^*}^R)^\infty) \le 2\cdot(10k+1)$. By Fact 9.5.5 and Claim 9.5.12, $\mathsf{Occ}'_{j^*} \supseteq \mathsf{Occ}_{j^*}$. The set Occ'_{j^*} is computed as follows in

this case: First, we identify p_{j^*} using Lemma 9.5.8. Then, we launch Ukkonen's online algorithm [279] to compute r_{j^*}, which takes $O(k^2)$ amortized time per character and $\tilde{O}(k^2)$ space. This procedure also returns an edit sequence ES'_{j^*} of length at most $2 \cdot (10k+1)$ between $T[p_{j^*}..p_{j^*} + r_{j^*} \cdot |Q_{j^*}|]$ and $(Q^R_{j^*})^\infty$.

Computing the distances. Consider the moment when $T[i]$ arrives. Let $i' = \lfloor i/2 \rfloor$ and $j^* = \max\{j : \ell_j + \lceil k/2 \rceil \leq i'\}$. If $i' + \ell_{j^*} \notin \mathsf{Occ}'_{j^*}$, the algorithm outputs $k+1$. Below we assume $i' + \ell_{j^*} \in \mathsf{Occ}'_{j^*}$.

If P_{j^*} is not $3k$-error periodic, we compute $e = \min_{j \in [i'-k..i'+k]} \mathsf{ed}_{\leq k}(T[..j], T(j..])$ by running k instances of Ukkonen's algorithm [279]. As $|\mathsf{Occ}'_{j^*}| = O(k^2)$, testing all positions in Occ'_{j^*} requires $\tilde{O}(k)$ space and $O(k^4)$ amortized time per character in total.

Otherwise, we compute e as follows. If $i' + \ell_{j^*} = p_{j^*}$, we use one instance of Ukkonen's algorithm [279] to compute an $O(k)$-length edit sequence ES''_{j^*} between $T[i'..i' + \ell_{j^*}]^R$ and $Q^\infty_j[1..\ell_{j^*}]$. For each $j \in [i'-k..i'+k]$, we first extract $O(k)$-edit sequences between $T[1..j]$ and $(Q_{j^*})^\infty[1..j]$ and between $T(j..i]$ and $(Q_{j^*})^\infty[1..j-i]$ from ES_{j^*}, ES'_{j^*}, and ES''_{j^*} in $O(k)$ time, and then apply Fact 9.5.6 and Lemma 9.5.7 to compute $\mathsf{ed}_{\leq k}(T[..j], T(j..])$ using $O(|Q_{j^*}|k^3)$ time and $O(k^2)$ space.

Summary. Computing $\mathcal{P}$ takes $\tilde{O}(k)$ amortized time per character and $\tilde{O}(k^2)$ space. The algorithm of Lemma 9.5.8 takes $\tilde{O}(k^4)$ amortized time per character and $\tilde{O}(k^4)$ space. Processing aperiodic prefixes requires $O(k^4)$ amortized time per character and $\tilde{O}(k)$ space. Processing periodic prefixes takes $O(k^4)$ time and $O(k^2)$ space as by the definition of Occ_j we test $O(k)$ positions out of $|Q_j|$.

Chapter 10

Small-space algorithms for Palindromic Length

10.1 Introduction

A *palindrome* is a non-empty string that equals its reversed copy, i.e., a string that reads the same both forward and backward. Throughout this work, we denote the language of palindromes by PAL, and the language of concatenations of k palindromes by $\mathsf{PAL}^k = \{P_1P_2\dots P_k : P_i \in \mathsf{PAL}, 1 \le i \le k\}$, for any $k \in \mathbb{N}^+$.

Recognizing PAL^k appeared to be and intricate problem. Galil and Seiferas [148] succeeded to design linear-time recognition algorithms for the cases $k = 1, 2, 3, 4$, but the general question remained open for almost 40 years. Only in 2015, Kosolobov et al. [216] showed an $O(nk)$-time recognition algorithm for PAL^k for all positive k, which was finally improved to the optimal $O(n)$ time by Rubinchik and Shur [257] in 2020. A related question is that of computing the *palindromic length* of a string T, which is defined to be the smallest integer k such that $T \in \mathsf{PAL}^k$. The first $O(n \log n)$-time algorithms for computing the palindromic length were presented in [133, 179, 256]. Finally, Borozdin, Kosolobov, Rubinchik, and Shur [76] showed an optimal $O(n)$-time algorithm for this problem.

Our contributions. In this work, we turn our attention to the *space complexity* of recognizing PAL^k and computing the palindromic length. We start by presenting a characterization of prefixes of a given string that belong to PAL^k. For $k = 1$, we refer to these prefixes as *prefix-palindromes*, and otherwise as *k-palindromic prefixes*. A crucial component of the linear time algorithm by Borozdin et al. [76] is the following well-known property: the prefix-palindromes of a length-n string can be expressed as $O(\log n)$ arithmetic progressions. If $x + a \cdot q$ with $a \in \{1, \dots, u\}$ is such a progression, then there are strings X of length x and Q of length q such that XQ^a is a prefix-palindrome, for every $a \in \{1, \dots, u\}$. The arithmetic progression can be encoded in $O(1)$ space, as it suffices to store x, q, and u.

In order to encode k-palindromic prefixes, we generalize arithmetic progressions to so-called *affine prefix sets* of order k. Intuitively, such a set consists of prefixes of the form $XQ_1^{a_1}Q_2^{a_2}\dots Q_k^{a_k}$ with $\forall i \in [1, k] : a_i \in \{1, \dots, u_i\}$. That is, rather than a single repeating substring Q, we allow multiple different substrings Q_i of different lengths. An affine prefix set of order k can then be encoded in $O(k)$ space. By carefully analyzing the rich structure of periodic substrings induced by k-palindromic prefixes, we show that the

k-palindromic prefixes can be expressed by a small number of affine prefix sets.

Theorem 10.1.1. *Let $0 < \epsilon < 1$ be constant. Let T be a string of length n and let $k \in \mathbb{N}^+$. The set of prefixes of T that belong to PAL^k is the union of $O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$ affine prefix sets, each of order at most k.*

Surprisingly, this representation is within polylog(n)-factors of the optimal encoding, at least for small values of k. We show the lower bound by explicitly constructing a large family of strings that can be uniquely identified by their palindromic prefixes:

Theorem 10.1.2. *Let T be a string of length n and let $k \in \mathbb{N}^+$. Encoding the lengths of the prefixes of T that belong to PAL^i, for each $i \in [1, k]$, requires $\Omega(k^{-k} \cdot (\log_3 n)^k)$ bits of space.*

As our final contribution, we derive a small-space read-only algorithm for constructing the affine prefix sets of Theorem 10.1.1. Particularly, we show how to compute a small-space representation of the i-palindromic prefixes of T for each $i \leq k$. Recall that, in the read-only model of computation, one has constant-time random access to the input string. The space complexity of the algorithm is the space used beyond storing the input string.

Theorem 10.1.3. *Let $0 < \epsilon < 1$ be constant. Given a string T of length n and $k \in \mathbb{N}^+$, there is a read-only algorithm that returns a compressed representation of all prefixes of T that belong to PAL^i, for each $i \in [1, k]$, in $O(n \cdot 6^{k^2/(2-\epsilon)} \cdot \log^k n)$ time and $O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$ space.*

As a corollary, we derive a parametrized read-only algorithm for computing the palindromic length.

Theorem 10.1.4. *Given a string T of length n, there is a read-only algorithm that computes the palindromic length k of T in $O(n \cdot 6^{k^2} \cdot \log^{\lceil k/2 \rceil} n)$ time and $O(6^{k^2} \cdot \log^{\lceil k/2 \rceil} n)$ space.*

In particular, for $k = O(\log \log n)$, the algorithm uses $n \log^{O(k)} n$ time and $\log^{O(k)} n$ space, and for $k = o(\sqrt{\log n})$, it uses $n^{1+o(1)}$ time and sublinear $n^{o(1)}$ space. In the regime of small palindromic length, this is an improvement over all previously-known algorithms [76, 257], which require $\Omega(n)$ space. It remains an intriguing open question whether it is possible to achieve *both* optimal linear time and sublinear space.

Note however that the lower bound of Theorem 10.1.2 does not imply a lower bound for our read-only algorithm as it has access to the input, and even more so for an algorithm that computes only the palindromic length of the input. On the other hand, proving an $\Omega(\log^{f(k)} n)$ space lower bound for a read-only algorithm might be well beyond the current techniques: the only lower bound technique for read-only string processing algorithms the authors are aware of is based on deterministic branching programs [75], and it shows that any read-only algorithm for computing the longest common substring of two strings that works in sublinear space must use slightly superlinear time; formally, an algorithm that uses $O(\tau)$ space requires $\Omega(n\sqrt{\log(n/\tau \log n)/\log\log(n/\tau \log n)})$ time [204].

Related work. Berenbrink, Ergün, Mallmann-Trenn, and Azer [60] initiated a study of the space complexity of computing the longest palindromic substring of a string in the streaming model, where the input arrives letter by letter and the space complexity is defined as the total space used, including any information an algorithm stores about the

input. They developed the first trade-offs between the bound on the error and the space complexity for approximating the length of the longest palindrome with either additive or multiplicative error, which were sequentially tightened by Gawrychowski, Merkurev, Shur, and Uznanski [161].

Amir and Porat [31] gave the first streaming algorithm for computing all approximate prefix-palindromes of a string. Namely, given an integer parameter k, their algorithm computes all prefixes within Hamming distance k from PAL. Bathie, Kociumaka, and Starikovskaya [49] improved their result for the Hamming distance and expanded it to the edit distance and the read-only setting (these results are presented in Chapter 9).

The problem of recognizing formal languages in small space has been also studied for regular languages, see [123, 150, 151, 152, 154], the Dyck language (the language of well-parenthesized expressions), see [185, 219, 232], for visibly pushdown languages (a language class strictly in-between the regular and context-free languages with good closure and decidability properties [27]), see [47, 142, 149], general context-free languages [153], and for DLIN and $\mathsf{LL}(k)$, see [41].

Roadmap. The remainder of the chapter is structured as follows. In Section 10.2, we introduce notation, basic definitions, and auxiliary lemmas. We then show the lower bound for encoding palindromic prefixes in Section 10.3. The space efficient encoding is presented in two steps. First, in Section 10.4, we describe affine prefix sets, their fundamental properties, and how they are related to the structure of periodic substrings. Then, in Section 10.5, we show how to encode the k-palindromic prefixes using affine prefix sets of order k, inductively assuming that the $(k-1)$-palindromic prefixes are already given as a union of affine prefix sets of order $k-1$. Finally, the algorithms from Theorem 10.1.3 and Theorem 10.1.4 are described in Section 10.6.

10.2 Preliminaries

Series, strings, and substrings. A series $a_1, b_1, c_1, a_2, b_2, c_2, \ldots, a_t, b_t, c_t$ is denoted by $(a_i, b_i, c_i)_{i=1}^{t}$. The empty series is denoted by ε. We use the dot-product to denote the concatenation of two series, e.g., $(a_i, b_i, c_i)_{i=1}^{t} = (a_i, b_i, c_i)_{i=1}^{t-3} \cdot (a_i, b_i, c_i)_{i=t-2}^{t}$. We may omit the subscript and superscript for series of length one, e.g., $(a_1, b_1, c_1) = (a_i, b_i, c_i)_{i=1}^{1}$.

In this chapter, we use the string-related definitions and notations introduced in Section 2.1, with the following modifications. We may call a substring $T[i..j]$ a *fragment* of T to emphasize that we mean the specific occurrence of $T[i..j]$ that starts at a position i. For example, in the string $T = \texttt{abcabc}$, the substrings $T[1..3]$ and $T[4..6]$ are identical, but $T[1..3]$ and $T[4..6]$ are distinct fragments of T. Furthermore, when introducing a string S, we may simply say "let $S[1..m]$ be a string" for "let S be a string of length m".

Rational powers of strings. In this chapter, we extend the notation of string powers to some rational numbers. Recall that for a non-negative integer m, the notation Q^m denotes the string obtained by concatenating m copies of Q. We extend this idea to non-negative rational exponents $\alpha \in \mathbb{Q}$, for which we write T^α to denote $T^{\lfloor \alpha \rfloor} \cdot T[1..(\alpha n \bmod n)]$. We only use this notation if $\alpha n \in \mathbb{N}$. For example, $(\texttt{abcdef})^{4/3} = \texttt{abcdefab}$.

Palindromes and periodicities. Recall that for a string $T[1..n]$, we write $\mathsf{rev}(T)$ to denote its reverse, i.e., $\mathsf{rev}(T) = T[n]T[n-1]\cdots T[1]$. We then say that T is a palindrome if and only if T is non-empty and $T = \mathsf{rev}(T)$. The set of all palindromes is denoted by

PAL. For a positive integer k, the set PAL^k contains all the strings that can be written as the concatenation of exactly k palindromes. We refer to such strings as *k-palindromic*; the smallest k such that T is k-palindromic is called the *palindromic length* of T. If a string P is a one-palindromic prefix of another string T, we also refer to it as *prefix-palindrome* of T.

A string $T[1..n]$ has *period* $p \in \mathbb{N}$ if $\forall i \in [1, n-p], T[i] = T[i+p]$, or equivalently if $T[1..n-p] = T(p..n]$. The string $T[1..n-p] = T(p..n]$ is a *border* of T. If T has period $p \leq n/2$, then we say that T is p-periodic. The smallest period of p is called *the* period of T. If T is p-periodic, then it can be written as $T = P^{\lfloor n/p \rfloor} P[1..n \bmod p]$, where $P = T[1..p]$. Alternatively, using the rational exponent, we have $T = P^{n/p}$.

We now provide some simple auxiliary lemmas regarding periodic string and palindromes.

Lemma 10.2.1. *For a primitive string Q, the period of Q^2 is $|Q|$.*

Proof. Let $q = |Q|$. If Q^2 has period $q' < q$, then Fact 2.1.1 implies that $p = \gcd(q', q) < q$ is a period of Q^2, hence also of Q. Then, however, it holds that $Q = Q[1..p]^{q/p}$, and q/p is an integer as p divides q, which contradicts the fact that Q is primitive. □

Lemma 10.2.2. *If a palindrome P has a q-periodic prefix of length $m \geq 3q/2$, then either P is q-periodic, or $|P| > 2m - q$.*

Proof. Since P is a palindrome, it does not only have a q-periodic prefix of length m, but also a q-periodic suffix of length m. If $|P| \leq 2m - q$, then the periodic prefix overlaps the periodic prefix by at least q letters, and hence it is easy to see that the entire P is q-periodic. □

Lemma 10.2.3. *Let Q be a string with suffix S and prefix P. For any positive integer x, the string SQ^xP is a palindrome if and only if $\mathsf{rot}^{|P|-|S|}(Q) = \mathsf{rev}(Q)$.*

Proof. Let $q = |Q|$, $s = |S|$, $p = |P|$, $\overline{Q} = \mathsf{rev}(Q)$, $\overline{S} = \mathsf{rev}(S)$, and $\overline{P} = \mathsf{rev}(P)$. We first assume $s \leq p$ and show that SQ^xP is a palindrome if and only if $\mathsf{rot}^{p-s}(Q) = \mathsf{rev}(Q)$. The individual steps are explained below.

$$\begin{aligned} \overline{P} \cdot \overline{Q}^x \cdot \overline{S} \;=\; S \cdot Q^x \cdot P \;&\iff_{(i)}\; \overline{P} \cdot \overline{Q}^x \;=\; S \cdot Q^x \cdot P[1..p-s] \\ &\iff_{(ii)}\; \overline{P} \cdot \overline{Q} \;=\; S \cdot Q[1..p-s] \cdot \mathsf{rot}^{p-s}(Q) \\ &\iff_{(iii)}\; \overline{Q} \;=\; \mathsf{rot}^{p-s}(Q). \end{aligned}$$

For (i), the $(\Rightarrow)$-direction is trivial, as we merely trim a suffix of length s on both sides. For the $(\Leftarrow)$-direction, $\overline{P} \cdot \overline{Q}^x = S \cdot Q^x \cdot P[..p-s]$ with $s \leq p$ implies that S is a prefix of $\overline{P}$, which means that $\overline{S}$ is a suffix of P. Hence $\overline{S} = P[p-s+1..p]$, and the $(\Leftarrow)$-direction of (i) follows. For step (ii), we use the fact that P is a prefix of Q. Finally, the $(\Rightarrow)$-direction of (iii) is trivial, as we trim a prefix of length p on both sides. For the $(\Leftarrow)$-direction, note that $\overline{Q} = \mathsf{rot}^{p-s}(Q)$ can be rewritten as

$$\overline{Q}[..q-p] \cdot \overline{P} = Q[p-s+1..q] \cdot Q[..p-s].$$

By trimming a prefix of length $q - p$ on both sides, we obtain

$$\overline{P} = Q[q-s+1..q] \cdot Q[..p-s] = SQ[..p-s].$$

Hence $\overline{Q} = \mathsf{rot}^{p-s}(Q)$ implies $\overline{P} \cdot \overline{Q} = SQ[..p-s] \cdot \mathsf{rot}^{p-s}(Q)$, and the $(\Leftarrow)$-direction of (iii) follows.

If $s > p$, then we simply invoke the lemma with $Q' = \overline{Q}$, which has suffix $S' = \overline{P}$ of length $s' = p$ and prefix $P' = \overline{S}$ of length $p' = s$ with $s' < p'$. We have already shown the correctness of the lemma for this case. Thus, $S'(Q')^x P' = \mathsf{rev}(SQ^x P)$ is a palindrome if and only if $\mathsf{rot}^{p'-s'}(Q') = \mathsf{rev}(Q')$, which is equivalent to $\mathsf{rot}^{-(s-p)}(Q) = Q$. □

Corollary 10.2.4. *Let y be a positive integer, let Q be a string, and let S and P be respectively a suffix and a prefix of Q^y. Let x be a positive integer, then $SQ^x P$ is a palindrome if and only if $\mathsf{rot}^{|P|-|S|}(Q) = \mathsf{rev}(Q)$.*

Proof. Let $q = |Q|$, $s = (|S| \bmod q)$, $p = (|P| \bmod q)$, $x' = \lfloor |S|/q \rfloor$ and $x'' = \lfloor |P|/q \rfloor$. Using these notations, we have $S = Q[q - s + 1..]Q^{x'}$ and $P = Q^{x''}Q[..p]$. Hence, by Lemma 10.2.3, $SQ^x P = Q[q - s + 1..]Q^{x'+x+x''}Q[1..]$ is a palindrome if and only if $\mathsf{rot}^{p-s}(Q) = \mathsf{rev}(Q)$. Note that $|P| - |S| = q \cdot (x'' - x') + p - s$, hence $|P| - |S| = p - s \pmod q$ and thus $\mathsf{rot}^{|P|-|S|}(Q) = \mathsf{rot}^{p-s}(Q)$, which concludes the proof. □

10.3 Lower Bound

We now show that the size of our representation of k-palindromic prefixes is close to optimal when k is small. For constant k, we are within an $O(\log n)$ factor of the optimal space (where the lower bound is $\Omega(\log^k n)$ *bits*, while our representation requires $O(\log^k n)$ *words*). For $k = O(\sqrt{\log \log n})$, we are within an $O(\mathrm{polylog}(n))$ factor of the lower bound.

Theorem 10.1.2. *Let T be a string of length n and let $k \in \mathbb{N}^+$. Encoding the lengths of the prefixes of T that belong to PAL^i, for each $i \in [1, k]$, requires $\Omega(k^{-k} \cdot (\log_3 n)^k)$ bits of space.*

For some string X and positive integer s, let $\mathsf{PalPref}^s(X) \subseteq [1, |X|] \times [1, s]$ be defined such that $(i, r) \in \mathsf{PalPref}^s(X)$ if and only if $r \leq s$ is the palindromic length of $X[..i]$. We will provide a lower bound for the space needed to encode $\mathsf{PalPref}^s(X)$. For this purpose, we construct for any integers $t, s \geq 1$ a family of strings $F(t, s)$ with the following properties:

1. for every $X \in F(t, s), |X| = 3^{t+s}$, and
2. we have $|F(t, s)| \geq 2^{b(t,s)}$, where $b(t, s) \geq \binom{t+s}{s}$, and
3. the function $f_s(X) = \mathsf{PalPref}^s(X)$ is injective on $F(t, s)$, and its inverse is computable.

Due to the second property, we can bijectively map all the possible bitstrings of length $b(t, s)$ to a subset of $F(t, s)$, and hence encoding an element of $F(t, s)$ requires at least $b(t, s)$ bits of space in the worst case. Assume that, for some string $X \in F(t, s)$, we are given the sets $\mathsf{PAL}^i(X)$ for every $i \leq s$. From these sets, we can easily construct $\mathsf{PalPref}^s(X)$. By the third property, we can then compute $X = f_s^{-1}(\mathsf{PalPref}^s(X))$. Hence the sets uniquely encode X, which implies that they cannot be stored in fewer than $b(t, s)$ bits of space. If we let $s = k$ and use $n = 3^{t+k}$ for any $t \geq 1$, this bound is at least $\binom{\log_3 n}{k} \geq ((\log_3 n)/k)^k$ bits.

Recursive construction of $F(t, s)$. For $t \geq 1$ and $s \geq 1$, we define

$$F(t, 1) = \{a^i b^{3^{t+1}-2i} a^i : i = 1, \ldots, 2^{t+1}\} \text{ where } a, b \text{ are distinct letters in } \Sigma, \text{ and}$$
$$F(1, s) = \{UVU : U, V \in F(1, s-1)\} \text{ when } s \geq 2, \text{ and}$$
$$F(t, s) = \{UVU : U \in F(t-1, s) \wedge V \in F(t, s-1)\} \text{ when } t, s \geq 2.$$

Note that $F(t,1)$ is well-defined, as $t \geq 1$, hence $2^{t+2} < 3^{t+1}$. In the above definition of $F(1,s)$ and $F(t,s)$, U and V are encoded on two disjoint copies of the alphabet, i.e. strings of $F(t,s)$ are defined over an alphabet of size 2^{s+t-1}.

Properties of $F(t,s)$. We now prove that $F(s,t)$ has the desired properties. The following claim can easily be proven by induction.

▷ Claim 10.3.1. For every $X \in F(s,t), |X| = 3^{t+s}$ and $X \in \mathsf{PAL}$.

▷ Claim 10.3.2. Let $b(t,s) = \log_2|F(t,s)|$. We have $b(t,s) \geq \binom{t+s}{s}$.

Claim proof. We proceed by induction. For $t \geq 1$, it holds $|F(t,1)| = 2^{t+1}$, i.e., $b(t,1) = t+1 = \binom{t+1}{1}$. In particular, $|F(1,1)| = 2^2$ and $b(1,1) = 2 = \binom{2}{1}$. For $s \geq 2$, it clearly holds $|F(1,s)| = |F(1,s-1)|^2$, and thus $b(1,s) = 2 \cdot b(1,s-1) \geq 2 \cdot \binom{s}{s-1} = 2s \geq s+1 = \binom{s+1}{s}$ by immediate induction. It remains the case where $t \geq 2$ and $s \geq 2$, in which we have $|F(t,s)| = |F(t-1,s)| \cdot |F(t,s-1)|$ and thus

$$b(t,s) = b(t-1,s) + b(t,s-1) \geq \binom{t+s-1}{s} + \binom{t+s-1}{s-1} = \binom{t+s}{s},$$

where the second step is by induction. Hence the claim holds. ◁

Finally, we show that every $X \in F(t,s)$ can be uniquely identified from $\mathsf{PalPref}^s(X)$.

▷ Claim 10.3.3. The function $f_s(X) = \mathsf{PalPref}^s(X)$ is injective on $F(t,s)$, and its inverse is computable.

Claim proof. In accordance with the recursive definition of $F(t,s)$, we proceed by induction. We start with the base case $s = 1$. The elements of $F(t,1)$ are of the form $X_i = a^i b^{3^{t+1}-2i} a^i$ for some $i \leq 2^{t+1}$. It follows that $(j,1) \in \mathsf{PalPref}^1(X_i)$ for all $j \leq i$, and $(i+1,1) \notin \mathsf{PalPref}^1(X_i)$. Hence f_1 is injective on $F(t,1)$, and we can compute the inverse $X_i = f_1^{-1}(\mathsf{PalPref}^1(X_i))$ due to $i = \min\{j \in \mathbb{N}^+ \mid (j+1,1) \notin \mathsf{PalPref}^1(X_i)\}$.

Next, assume that $t = 1$ and $s \geq 2$, and we inductively assume that f_{s-1} is injective on $F(1,s-1)$. Let $X, Y \in F(1,s)$, and assume that $\mathsf{PalPref}^s(X) = \mathsf{PalPref}^s(Y)$. Note that there exist strings $U, U', V, V' \in F(1,s-1)$ such that $X = UVU$ and $Y = U'V'U'$. We show $U = U'$ and $V = V'$, which implies $X = Y$. Note that, as U is a prefix of X, $\mathsf{PalPref}^{s-1}(U)$ can be computed from $\mathsf{PalPref}^s(X)$ using

$$\mathsf{PalPref}^{s-1}(U) = \mathsf{PalPref}^s(X) \cap ([1..|U|] \times [1..s-1]).$$

The same equation holds for U' and Y, hence $\mathsf{PalPref}^{s-1}(U) = \mathsf{PalPref}^{s-1}(U')$. As inductively assume that $f_{s-1}(X) = \mathsf{PalPref}^{s-1}(X)$ is injective on $F(1,s-1)$, it follows that $U = U'$.

We now turn to showing that $V = V'$. Consider an index $i \in [|U|+1..|UV|]$, and let s' be the palindromic length of $X[..i] = U \cdot V[..i-|U|]$. By definition, $X[..i]$ can be decomposed into s' palindromes. Since U and V are encoded over distinct alphabets, any palindrome in this decomposition is fully contained in U or in $V[..i-|U|]$, therefore $V[..i-|U|]$ can be decomposed into some number $s'' < s'-1$ of palindromes. By minimality of s', the palindromic length of $V[..i-|U|]$ is at least $s'-1$, hence the palindromic length of $V[..i-|U|]$ is exactly $s'-1$. Therefore, $\mathsf{PalPref}^{s-1}(V)$ can also be derived from $\mathsf{PalPref}^s(X)$:

$$\mathsf{PalPref}^{s-1}(V) = \{(i-|U|, s'-1) : (i,s') \in \mathsf{PalPref}^s(X)\}.$$

As the same relation holds for V' and Y, it follows that $\mathsf{PalPref}^{s-1}(V) = \mathsf{PalPref}^{s-1}(V')$, and by induction hypothesis, this implies that $V = V'$, and therefore $X = Y$. The inverse function f_s^{-1} can be efficiently computed using two recursive calls to f_{s-1}^{-1}.

The proof for the case when $t, s \geq 2$ is highly similar, but the induction is over the sum $t+s$. We inductively assume that f_{s-1} is injective on $F(t, s-1)$, and that f_s is injective on $F(t-1, s)$. Instead of $\mathsf{PalPref}^{s-1}(U)$, we consider $\mathsf{PalPref}^{s}(U)$. The remainder of the proof is the same as for $t = 1$ and $s \geq 2$. ◁

This concludes the proof of Theorem 10.1.2.

10.4 Combinatorial Properties of Affine Prefix Sets

In this section, we study the combinatorial structure of k-palindromic prefixes of T. We start with the definition of *affine sets*, which we will use as a scaffolding for our analysis.

Definition 10.4.1 (Affine sets)**.** *A set of strings $\mathcal{A}$ is* affine *if there exist $t \in \mathbb{N}_0$, a string X, primitive strings $Q_1, \ldots, Q_t$, and positive integers $\ell_1, \ldots, \ell_t$ and $u_1, \ldots, u_t$ such that*

$$\forall i \in [1, t] : \ell_i \leq u_i \qquad \text{and} \qquad \mathcal{A} = \{XQ_1^{a_1} \ldots Q_t^{a_t} \mid \forall r \in [1, t] : a_r \in [\ell_r, u_r]\}.$$

The tuple $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$ is a representation *of $\mathcal{A}$, and t is the* order *of the representation. The order of $\mathcal{A}$ is the minimal order achieved by any of its representations. We call $\{Q_i\}$* the components *of a representation, and ℓ_i (resp., u_i) the exponent lower (resp., upper) bounds.*

A representation *generates* (the strings of) the corresponding affine string set. If $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$ generates $\mathcal{A}$ and $\langle X', (Q'_i, \ell'_i, u'_i)_{i=1}^{t'} \rangle$ generates $\mathcal{B}$, then their concatenation is defined as $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \cdot (Y, a, a) \cdot (Q'_i, \ell'_i, u'_i)_{i=1}^{t'} \rangle$, where Y is a primitive string and a is a positive integer such that $Y^a = X'$ (i.e., Y is the primitive root of X'). The concatenation generates $\mathcal{A} \cdot \mathcal{B} = \{A \cdot B : A \in \mathcal{A} \wedge B \in \mathcal{B}\}$. (If $X' = \varepsilon$, then the concatenation is $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \cdot (Q'_i, \ell'_i, u'_i)_{i=1}^{t'} \rangle$.)

In what follows, we consider *affine prefix sets*, i.e., affine sets containing only prefixes of the given input string T. We will show that a small number of affine prefix sets suffices to represent the k-palindromic prefixes of T. An example for $k = 2$ is provided in Fig. 10.1. The case where $k = 1$, i.e., the structure of prefix-palindromes, is well-understood: there are $O(\log n)$ groups of such palindromes, where each group can be expressed as an arithmetic progression and a corresponding periodic prefix of T (see e.g. [76, Lemma 5]). Below, we restate this result in the framework of affine prefix sets (with proofs provided merely for self-containedness).

Lemma 10.4.2. *Let S be a palindrome, and let P be the longest proper prefix of S that is a palindrome. If $|S| \leq 3|P|/2$, then there exist strings $U \in \mathsf{PAL} \cup \{\varepsilon\}$, $V \in \mathsf{PAL}$ and an integer $i \geq 3$ such that $P = U(VU)^{i-1}$ and $S = U(VU)^i$. Furthermore, VU is primitive, and $|VU|$ is the minimal period of both S and P.*

Proof. Let Q' be such that $S = PQ'$, and let Q be the primitive root of Q', i.e., $Q' = Q^h$ for some positive integer h. As $|S| \leq 3|P|/2$, we have $|Q'| \leq |P|/2$. Now, since P is a prefix-palindrome of S and S is a palindrome, $\mathsf{rev}(P) = P$ is a suffix of S. In other words, S has a border $|P|$, and $|Q'|$ is a period of S. Since S has suffix Q', the periodicity implies that S is a suffix of $Q^{ri'} = Q^{hi'}$ for some positive integer i'. Therefore, it holds $S = UQ^i$

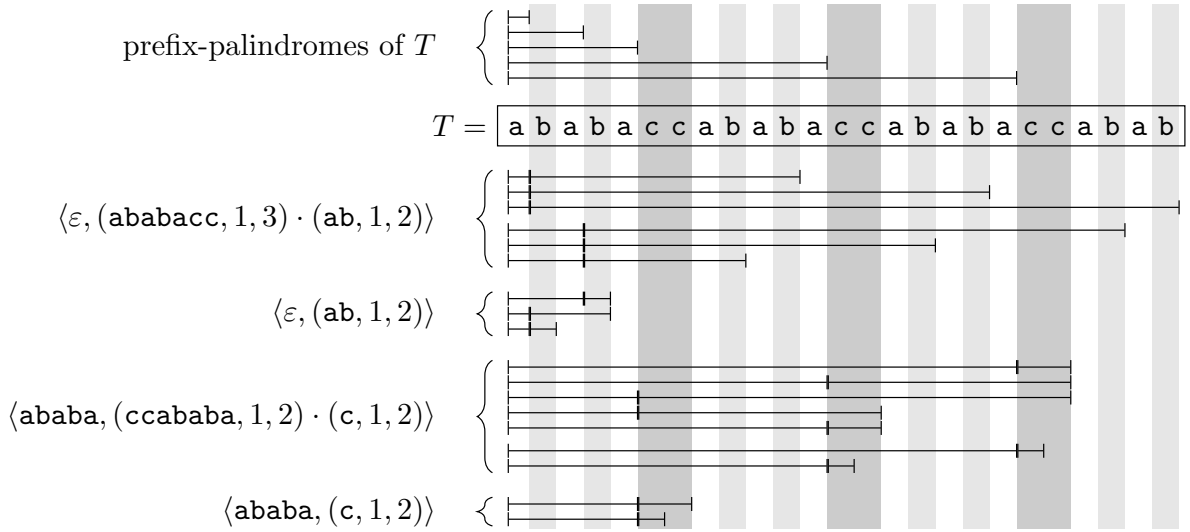

Figure 10.1: String T with prefix-palindromes `a`, `aba`, `ababa`, `ababaccababa`, and `ababaccababaccababa`. The prefix-palindromes can be expressed using four affine prefix sets with representations $\langle \mathtt{a}, \varepsilon \rangle$, $\langle \mathtt{aba}, \varepsilon \rangle$, $\langle \mathtt{a}, (\mathtt{ba}, 1, 2) \rangle$, and $\langle \mathtt{ababa}, (\mathtt{ccababa}, 1, 2) \rangle$. The same number of affine prefix sets suffices to express the prefixes of T in PAL^2, as shown below T. There are 14 such prefixes, and some of them can be split into two palindromes in multiple ways.

for some integer $i \geq h$ and a proper suffix U of Q. This implies $P = UQ^{i-h}$ because of $S = PQ^h$.

Let V be such that $Q = VU$, then S has a prefix UV and a suffix VU. Since S is a palindrome, it holds $UV = \mathsf{rev}(VU) = \mathsf{rev}(U)\mathsf{rev}(V)$. We know that V is non-empty because U is a proper suffix of Q, thus $V \in \mathsf{PAL}$ and $U \in \mathsf{PAL} \cup \{\varepsilon\}$.

Let $j = i - h + 1$. Let $P' = U(VU)^j = (UV)^j U = (\mathsf{rev}(U)\mathsf{rev}(V))^j \mathsf{rev}(U) = \mathsf{rev}(U(VU)^j)$ and note that P' is a palindrome of length $|PQ|$. If $h > 1$, i.e., if Q' is non-primitive, then P' is a proper prefix of S, which contradicts the fact that P is the longest palindromic prefix of S. Hence $h = 1$ and $Q' = Q = VU$ is primitive. If $i \leq 2$ then $|Q| \geq |P|/2$, but we have already shown $|Q'| \leq |P|/2$. Hence $i \geq 3$. Finally, both P and S have suffix Q^2 and period $|Q|$. By Lemma 10.2.1, $|Q|$ is the minimal period of Q^2, and thus also of P and S. □

Corollary 10.4.3. *The prefix-palindromes of a string T of length n can be partitioned into $O(\log n)$ affine sets of order at most 1. Each set $\mathcal{S}_i$ of order 1 has a representation of the form $\langle U_i(V_iU_i)^{\ell_i}, (V_iU_i, 1, u_i) \rangle$ for some $U_i \in \mathsf{PAL} \cup \{\varepsilon\}$, $V_i \in \mathsf{PAL}$ and integers $\ell_i \geq 1$ and $u_i > 1$.*

Proof. Let $P_1, \ldots, P_m$ denote the prefix-palindromes of T, ordered by increasing lengths. We iterate over the prefix-palindromes starting with P_2, maintaining an active affine set $\mathcal{A}$ that is initially represented by $\langle T[1], \varepsilon \rangle$ of order 0 (because every string T has shortest prefix-palindrome $T[1]$). When processing P_i, we consider two cases:

a) If $|P_i| > 3|P_{i-1}|/2$, we create a new affine set $\mathcal{A}$ containing only P_i, represented by $\langle P_i, \varepsilon \rangle$ of order 0. This set will become the new active affine set.
b) If $|P_i| \leq 3|P_{i-1}|/2$, then we add P_i to the active set using one of two subcases.
   i) If the representation of the active set is $\langle P_{i-1}, \varepsilon \rangle$, then we add P_i to the active set. By Lemma 10.4.2, there are $U \in \mathsf{PAL} \cup \{\varepsilon\}$, $V \in \mathsf{PAL}$ and integer $j \geq 3$

such that $P_{i-1} = U(VU)^{j-1}$ and $P_i = U(VU)^j$. Furthermore, VU is primitive, and $|VU|$ is the minimal period of P_{i-1} and P_i. We replace $\langle P_i, \varepsilon\rangle$ with $\langle U(VU)^{j-2}, (VU, 1, 2)\rangle$.

ii) Otherwise, we can inductively assume the following invariant. The active set has representation $\langle U(VU)^\ell, (VU, 1, u)\rangle$ for integers $\ell \geq 1$ and $u > 1$, where $|VU|$ is the minimal period of the most recently added palindrome $P_{i-1} = U(VU)^{\ell+u}$. Clearly, the invariant holds if P_{i-1} was processed using Case (b.i), and we will ensure that it also holds after using Case (b.ii).
By Lemma 10.4.2, P_i and P_{i-1} have the same minimal period, which is $|VU|$ due to the invariant. Also, the lemma implies $P_i = P_{i-1}[..|VU|]P_{i-1}$, which is $U(VU)^{\ell+u+1}$ due to the invariant. Hence we can add P_i to the active set by merely increasing u in the representation by one, which clearly maintains the invariant.

We then proceed with the next palindrome P_{i+1}. In Case (a)), the length of the current prefix-palindrome exceeds the length of the previous one by a factor of 3/2. Hence this case occurs at most $\lceil \log_{3/2} n \rceil$ times, resulting in $O(\log n)$ affine sets. In Case (b)), we do not create any affine sets. The correctness follows from the description of the cases. In particular, as seen in Case (b)), the representations of order 1 satisfy the stated properties. □

10.4.1 Reducing affine prefix sets

In what follows, all affine sets that we consider are affine prefix sets, therefore the term "affine set" implicitly means "affine prefix set". A single affine set may have multiple equivalent representations. For example, the affine set $S = \{\texttt{caba}, \texttt{cababa}, \texttt{cabababa}\}$ is represented by $\langle \texttt{c}, (\texttt{ab}, 1, 3), (\texttt{a}, 1, 1)\rangle$ and $\langle \texttt{ca}, (\texttt{ba}, 1, 3)\rangle$. Arguably, the latter representation is preferable, as it has a lower order and can thus be encoded more efficiently. Hence we propose a way of potentially decreasing the order of a representation by *reducing* it.

Definition 10.4.4 (Irreducible representation)**.** *A representation $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ of an affine string set is* irreducible *if:*

- *for every $r \in [1..t]$, $\ell_r = 1 < u_r$, and*
- *for every $r \in [1..t)$, we have $|Q_r| > |Q_{r+1}|$.*

From now on, we say that Q_r with $r \in [1, t]$ is *fixed* if $\ell_r = u_r$, and *flexible* otherwise. If there is some $r \in [1, t)$ such that $|Q_r| \leq |Q_{r+1}|$, then we say that there is an *inversion* between Q_r and Q_{r+1}. Thus, a representation is irreducible if and only if all components are flexible and have unit lower bounds, and there are no inversions. As per this definition, $\langle \texttt{ca}, (\texttt{ba}, 1, 3)\rangle$ is the only irreducible representation of S from the previous example. Another example is provided in Fig. 10.2.

Properties of flexible components. Now we show how to make an arbitrary representation irreducible, possibly decreasing (but never increasing) its order. The reduction exploits the structure of periodic substrings induced by flexible components.

Lemma 10.4.5. *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be a representation of an affine prefix set, and consider any $r \in [1, t)$ such that Q_r is flexible. Then $|Q_r|$ is a period of every string $Q_r^{a_r} Q_{r+1}^{a_{r+1}} \dots Q_t^{a_t}$ that satisfies $a_r \in \mathbb{N}_0$ and $\forall j \in (r, t] : a_j \in [\ell_j, u_j]$.*

Proof. Let $P = Q_1^{\ell_1} Q_2^{\ell_2} \dots Q_r^{\ell_r}$ and $S = Q_{r+1}^{a_{r+1}} Q_{r+2}^{a_{r+2}} \dots Q_t^{a_t}$. By the definition of an affine prefix set, XPS is a prefix of the underlying string T. Since Q_r is flexible, it holds $\ell_r < u_r$,

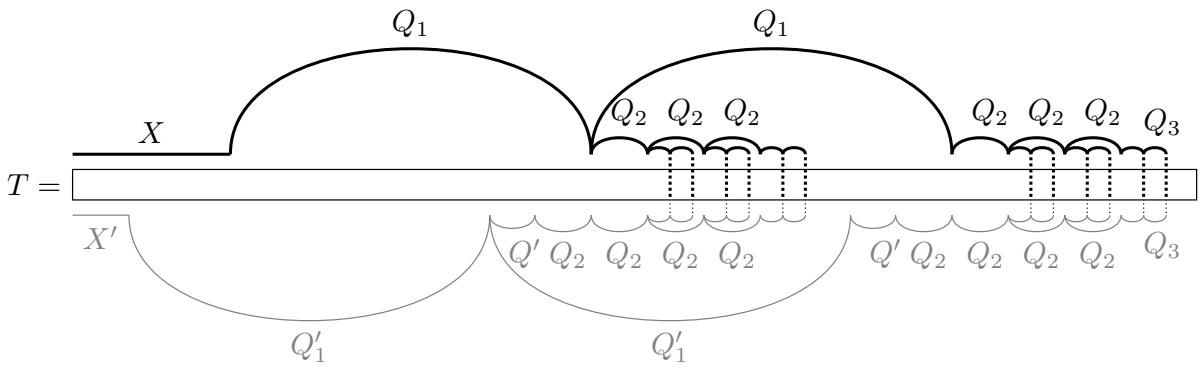

Figure 10.2: An affine prefix set $\mathcal{A}$ of a string T with representation $\langle X, (Q_1, 1, 2) \cdot (Q_2, 1, 3) \cdot (Q_3, 1, 2)\rangle$ (drawn above T). This representation is irreducible. The set $\mathcal{A}$ contains all the prefixes of T that end at positions drawn in dotted lines. In this example, the set $\mathcal{A}$ has the alternative representation $\langle X', (Q'_1, 1, 2) \cdot (Q', 1, 1) \cdot (Q_2, 2, 4) \cdot (Q_3, 1, 2)\rangle$. This representation is reducible because Q' has the same exponent upper and lower bound, and because Q_2 has an exponent lower bound larger than 1.

and thus XPQ_rS is also a prefix of T. If both XPS and XPQ_rS are prefixes of T, then S is a prefix of Q_rS. Hence Q_rS has border S and period $|Q_r|$. If Q_rS has period $|Q_r|$, then clearly Q_r^aS, for all $a \in \mathbb{Z}^+$, also has period $|Q_r|$. □

If two adjacent components Q_r and Q_{r+1} are flexible, then the lemma allows us to obtain the following lower bound on the length of Q_r.

Lemma 10.4.6. *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be a representation of an affine prefix set, and let $r \in [1, t)$. If both Q_r and Q_{r+1} are flexible, then either $Q_r = Q_{r+1}$ or*

$$|Q_r| > |Q_{r+1}^{u_{r+1}-1}| + \left(\sum_{j=r+2}^{t} |Q_j^{u_j}|\right) + \gcd(|Q_r|, |Q_{r+1}|).$$

Proof. For flexible Q_r and Q_{r+1}, let $q_r = |Q_r|$, $q_{r+1} = |Q_{r+1}|$, and $p = \gcd(q_r, q_{r+1})$. Let $Q'_{r+1} = Q_{r+1}^{u_{r+1}} Q_{r+2}^{u_{r+2}} \dots Q_t^{u_t}$. By Lemma 10.4.5, both q_r and q_{r+1} are periods of Q'_{r+1}, and q_r is a period of $Q_rQ'_{r+1}$. Since q_r is a period of $Q_rQ'_{r+1}$, it is also a period of Q_rQ_{r+1}. Hence $Q_r = Q_{r+1}$ if and only if $q_r = q_{r+1}$. For the sake of contradiction, assume that the lemma does not hold, i.e., $q_r \neq q_{r+1}$ and $q_r \leq |Q'_{r+1}| - q_{r+1} + p$. We make two observations.

First, Q'_{r+1} is of length $|Q'_{r+1}| \geq q_r + q_{r+1} - p$, and it has distinct periods q_r and q_{r+1}. The Periodicity Lemma (Fact 2.1.1) implies that p is a period of Q'_{r+1}, and thus also a period of its prefix Q_{r+1}. If $p < q_{r+1}$, then $Q_{r+1} = Q_{r+1}[1..p]^{q_{r+1}/p}$, which contradicts the primitivity of Q_{r+1}. Second, Q'_{r+1} is of length $|Q'_{r+1}| \geq q_r + q_{r+1} - p \geq q_r$. Since q_r is a period of $Q_rQ'_{r+1}$, we know that Q_r is a prefix of Q'_{r+1}. Hence p is also a period of Q_r. If $p < q_r$, then $Q_r = Q_r[1..p]^{q_r/p}$, which contradicts the primitivity of Q_r.

We have shown that $\gcd(q_r, q_{r+1}) \geq \max(q_r, q_{r+1})$. This is only possible if the gcd of q_r and q_{r+1} is equal to both q_r and q_{r+1}, which contradicts the assumption that $q_r \neq q_{r+1}$. Therefore, the lemma holds. □

Lemma 10.4.7. *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be an irreducible representation of an affine prefix set $\mathcal{A}$ of a string of length n. Then it holds $|\mathcal{A}| = \prod_{i=1}^t u_i$ and $t \leq \log_2 n$.*

Proof. Let $E = \{(a_i)_{i=1}^t \mid \forall i \in [1,t] : a_i \in [1,u_i]\}$ of cardinality $|E| = \prod_{i=1}^t u_i$ be the set of exponent configurations admitted by the representation (where $[1,u_i] = [\ell_i,u_i]$ because the representation is irreducible). Then $\mathcal{A} = \{XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t} \mid (a_i)_{i=1}^t \in E\}$. In order to show $|\mathcal{A}| = |E|$, it suffices to show that no two elements in E generate the same string.

For the sake of contradiction, assume that there are distinct sequences $(a_i)_{i=1}^t, (b_i)_{i=1}^t \in E$ that generate the same string $S = XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t} = XQ_1^{b_1}Q_2^{b_2}\dots Q_t^{b_t}$. Let $r \in [1,t]$ be the minimal index such that $a_r \neq b_r$, and assume w.l.o.g. that $a_r > b_r$. Then S has the prefix $XQ_1^{a_1}\dots Q_{r-1}^{a_{r-1}}Q_r^{b_r} = XQ_1^{b_1}\dots Q_{r-1}^{b_{r-1}}Q_r^{b_r}$, and we can factorize the corresponding suffix in two different ways as $Q_r^{a_r-b_r}Q_{r+1}^{a_{r+1}}\dots Q_t^{a_t} = Q_{r+1}^{b_{r+1}}\dots Q_t^{b_t}$. However, the two factorizations have different lengths $|Q_r^{a_r-b_r}Q_{r+1}^{a_{r+1}}\dots Q_t^{a_t}| > |Q_rQ_{r+1}| > |Q_{r+1}^{u_{r+1}}\dots Q_t^{u_t}| \geq |Q_{r+1}^{b_{r+1}}\dots Q_t^{b_t}|$, where the second inequality is due to Lemma 10.4.6. Because of this contradiction, there cannot be distinct sequences $(a_i)_{i=1}^t, (b_i)_{i=1}^t \in E$ that define the same string.

Finally, it holds $\forall i \in [1,t] : u_i \geq 2$ for any irreducible representation, and thus $|\mathcal{A}| = \prod_{i=1}^t u_i \geq 2^t$. Since trivially $|\mathcal{A}| \leq n$, it follows $2^t \leq n$ or equivalently $t \leq \log_2 n$. □

Transforming representations. Now we use the properties of flexible components to transform an arbitrary representation into an irreducible one. We use the following operations.

Lemma 10.4.8. *Let $\rho = \langle X, (Q_i,\ell_i,u_i)_{i=1}^t\rangle$ be a representation of an affine prefix set.*

1. *If there is $r \in [1,t)$ such that Q_r is flexible and Q_{r+1} is fixed, then let $x = |Q_{r+1}^{\ell_{r+1}}| \bmod |Q_r|$. The affine prefix set has representation*

   $$\mathsf{switch}_r(\rho) = \langle X, (Q_i,\ell_i,u_i)_{i=1}^{r-1} \cdot (Q_{r+1},\ell_{r+1},u_{r+1}) \cdot (\mathsf{rot}^x(Q_r),\ell_r,u_r) \cdot (Q_i,\ell_i,u_i)_{i=r+2}^t\rangle.$$

2. *If there is $r \in [1,t)$ such that such that both Q_r and Q_{r+1} are flexible and $|Q_r| \leq |Q_{r+1}|$, then $Q_r = Q_{r+1}$ and the affine prefix set has representation*

   $$\mathsf{merge}_r(\rho) = \langle X, (Q_i,\ell_i,u_i)_{i=1}^{r-1} \cdot (Q_r,\ell_r+\ell_{r+1},u_r+u_{r+1}) \cdot (Q_i,\ell_i,u_i)_{i=r+2}^t\rangle.$$

3. *If there is $r \in [1,t]$ such that $\ell_r > 1$, then the affine prefix set has representation*

   $$\mathsf{split}_r(\rho) = \langle X, (Q_i,\ell_i,u_i)_{i=1}^{r-1} \cdot (Q_r,\ell_r-1,\ell_r-1) \cdot (Q_r,1,u_r-\ell_r+1) \cdot (Q_i,\ell_i,u_i)_{i=r+1}^t\rangle.$$

4. *If Q_1 is fixed, then the affine prefix set has representation*

   $$\mathsf{truncate}(\rho) = \langle XQ_1^{\ell_1}, (Q_{i+1},\ell_{i+1},u_{i+1})_{i=1}^{t-1}\rangle.$$

Proof. Statements (3) and (4) are trivial. For (2), if $|Q_r| \leq |Q_{r+1}|$ and both Q_r and Q_{r+1} are flexible, then Lemma 10.4.6 implies $Q_r = Q_{r+1}$. Hence the statement follows.

Finally, we show that statement (1) holds. Assume that Q_r is flexible and Q_{r+1} is fixed. Then Lemma 10.4.5 implies that $|Q_r|$ is a period of $Q_rQ_{r+1}^{\ell_{r+1}}$, and thus $Q_{r+1}^{\ell_{r+1}} = Q_r^xQ_r[1..y]$ with $x = \lfloor |Q_{r+1}^{\ell_{r+1}}|/|Q_r| \rfloor$ and $y = |Q_{r+1}^{\ell_{r+1}}| \bmod |Q_r|$. (Either x or y might be zero, but this is irrelevant for the proof.) Let $P = Q_r[1..y]$ and $S = Q_r(y..|Q_r|]$. Any rotation of a primitive string is primitive, and hence $\mathsf{rot}^y(Q_r) = SP$ is primitive. For any exponent $a \in [\ell_r,u_r]$, it holds $Q_r^aQ_{r+1}^{\ell_{r+1}} = (PS)^a(PS)^xP = (PS)^xP(SP)^a = Q_{r+1}^{\ell_{r+1}}(\mathsf{rot}^y(Q_r))^a$. Hence the stated transformation does not change the represented affine prefix set. □

The leftmost (i.e., lowest index) fixed component Q_r of a representation can either be removed with $\mathsf{truncate}$ (if $r = 1$), or it can be moved further to the left with switch_{r-1} (if $r > 1$). By repeatedly applying $\mathsf{truncate}$ and switch, we obtain the following lemma.

Lemma 10.4.9. *Let $\rho = \langle X, (Q_i, \ell_i, u_i)_{i=1}^{t} \rangle$ be a representation of an affine prefix set, and let $F = \{j \in [1,t] \mid \ell_j < u_j\} = \{j_1, \dots, j_{|F|}\}$ with $j_1 < j_2 < \cdots < j_{|F|}$ be the indices of the flexible components. Then the affine prefix set has a representation $\langle \hat{X}, (\hat{Q}_{j_i}, \ell_{j_i}, u_{j_i})_{i=1}^{|F|} \rangle$ such that $\hat{Q}_{j_i}$ is a rotation of Q_{j_i} for every $i \in [1, |F|]$. Both $\hat{X}$ and all the $\hat{Q}_{j_i}$ are functions of X, $Q_1, \dots, Q_t$, and $\ell_1, \dots, \ell_t$, i.e., they are independent of $u_1, \dots, u_t$.*

Proof. We transform ρ by repeatedly applying Lemma 10.4.8. First, as long as there is some flexible component Q_r that is followed by a fixed component Q_{r+1}, we apply $\rho \leftarrow \mathsf{switch}_r(\rho)$. Conceptually speaking, this moves fixed components further to the left and flexible components further to the right (without changing the order of the representation). Hence it is easy to see that the procedure terminates. Afterwards, any component Q_r is fixed if and only if $r \in [1, t - |F|]$. Now we merge all the fixed components into X by applying $\rho \leftarrow \mathsf{truncate}(\rho)$ exactly $t - |F|$ times. This results in a representation of order $|F|$ (possibly 0) in which all components are flexible.

Whenever switch_r produces a rotation of one of the primitive strings, the outcome depends solely on the lengths and exponent lower bounds of the participating components, while the exponent upper bounds are irrelevant. Similarly, $\mathsf{truncate}$ modifies X according to its current value, as well as the length and exponent lower bound of the current leftmost component. Hence $\hat{X}$ and all the $\hat{Q}_{j_i}$ are indeed independent of the exponent upper bounds. □

After applying Lemma 10.4.9, we repeatedly apply merge to remove all inversions. Then, we apply split until all flexible components have exponent lower bound 1. This may result in new fixed components, which we remove with Lemma 10.4.9, resulting in the following lemma.

Lemma 10.4.10. *An affine prefix set represented by $\rho = \langle X, (Q_i, \ell_i, u_i)_{i=1}^{t} \rangle$ has an irreducible representation of order $|L| \le t$, where $L = \{|Q_r| \mid r \in [1,t] : \ell_r < u_r\}$ is the set of distinct* lengths *of flexible components.*

Proof. By applying Lemma 10.4.9 to ρ, we obtain a representation $\rho' = \langle X, (Q'_i, \ell'_i, u'_i)_{i=1}^{t'} \rangle$ of order $t' = |\{i \in [1,t] \mid \ell_i < u_i\}|$ that contains only flexible components. Note that Lemma 10.4.9 implies that the components of this representation still have lengths in $L = \{|Q'_r| \mid r \in [1,t']\}$. Consider any length $q \in L$, and let $r_{\min}, r_{\max} \in [1,t']$ be the respectively minimal and maximal index such that $|Q'_{r_{\min}}| = |Q'_{r_{\max}}| = q$. Then Lemma 10.4.6 implies that for every y in the interval $[r_{\min}, r_{\max}]$, we have $Q'_r = Q'_{r_{\min}}$. We apply $\rho' \leftarrow \mathsf{merge}_{r_{\min}}(\rho')$ exactly $r_{\max} - r_{\min}$ times and merge all the components of length q into a single one. By applying this procedure for each length $q \in L$, we obtain a representation of $|L|$ components.

Now ρ' is inversion-free, contains only flexible components, and is of order $|L|$. As long as there is a flexible component Q_r such that $\ell_r > 1$, we apply $\rho \leftarrow \mathsf{split}_r(\rho)$. This creates at most one additional fixed component for each of the $|L|$ flexible components. We remove the fixed components using Lemma 10.4.9, leaving the number of flexible components, their exponent lower bounds, and their lengths unchanged. Hence we obtain the final irreducible representation of order $|L|$. □

10.4.2 Strongly affine representations

Later, we will describe and exploit intricate properties of repetitive fragments induced by affine prefix sets. These properties are easier to show if we can assume that the periodicity can be extended beyond the considered region by a constant number of additional

repetitions of the period. Consequently, we introduce the notion of a *strongly affine representation* of an affine prefix set of T, which corresponds to representations in which the exponent upper bound of every (flexible) component can be increased by five and still yield an affine prefix set of T. This is visualized in Fig. 10.3.

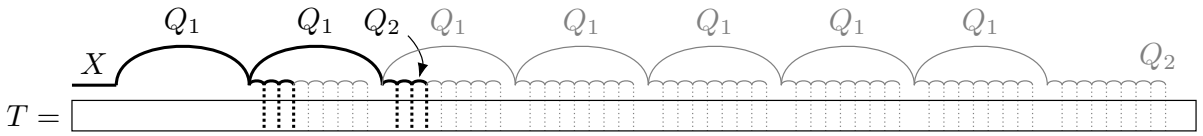

Figure 10.3: An affine prefix set $\mathcal{A}$ of a string T with representation $\langle X, (Q_1, 1, 2) \cdot (Q_2, 1, 3)\rangle$ (drawn in black). If this representation is strongly affine, then its expansion $\langle X, (Q_1, 1, 7) \cdot (Q_2, 1, 8)\rangle$ is also a representation of an affine prefix set of T (drawn in gray).

Definition 10.4.11 (Strongly affine representations)**.** *A representation*

$$\rho = \langle X, (Q_i, \ell_i, u_i)_{i=1}^{t}\rangle$$

of an affine prefix set of a string T is strongly affine *if and only if its* periodic expansion

$$\mathsf{expand}(\rho) = \langle X, (Q_i, \ell_i, u'_i)_{i=1}^{t}\rangle \text{ with } \forall i \in [1, t] : \begin{cases} u'_i = u_i & \text{if } u_i = \ell_i \\ u'_i = u_i + 5 & \text{otherwise} \end{cases}$$

is also the representation of an affine prefix set of T.

Definition 10.4.12 (Canonical representation)**.** *A representation of an affine prefix set is* canonical *if and only if it is both strongly affine and irreducible.*

It can be readily verified that, if ρ is strongly affine, then $\mathsf{truncate}(\rho)$, $\mathsf{split}_r(\rho)$, $\mathsf{merge}_r(\rho)$, and $\mathsf{switch}_r(\rho)$ are also strongly affine (for any r, assuming that the respective operation is indeed applicable). We obtained Lemma 10.4.10 by applying a sequence of these operations, and hence we have the following immediate corollary.

Corollary 10.4.13. *An affine prefix set with strongly affine representation*

$$\rho = \langle X, (Q_i, \ell_i, u_i)_{i=1}^{t}\rangle$$

has a canonical representation of order $|L| \le t$, where $L = \{|Q_r| \mid r \in [1, t] : \ell_r < u_r\}$ is the set of distinct lengths of flexible components.

Whether a representation ρ of an affine prefix set $\mathcal{A}$ of T is strongly affine does not only depend on ρ, it also depends on what T looks like beyond the end of the longest prefix represented by ρ. Therefore, one cannot hope to transform an arbitrary representation into an equivalent strongly affine representation. However, by "removing" the last five copies of each component and treating them separately, we show that we can cover an affine prefix set of order t with at most 6^t canonical representations.

Lemma 10.4.14. *An affine prefix set of order t can be partitioned into 6^t affine prefix sets, each of which has a canonical representation of order at most t.*

Proof. Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$ be a representation of an affine prefix set. We produce a set of representations defined by

$$R = \left\{ \langle X, (Q_i, \ell'_i, u'_i)_{i=1}^t \rangle : \forall r \in [1,t], (\ell'_r, u'_r) \in B_r \right\},$$
$$\text{where } \forall r \in [1,t] : B_r = \{(u,u) \mid u \in [\max(\ell_r, u_r - 4), u_r]\} \cup \{(\ell_r, \max(\ell_r, u_r - 5))\}. \tag{10.1}$$

It is easy to see that the affine sets generated by representations in R form a partition of the affine set generated by $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$. By design, for any representation in R, and for any component Q_r, we know that Q_r is either fixed, or it has exponent lower bound ℓ_r and exponent upper bound $u_r - 5$. Hence the instances in R are strongly affine, and it follows from Corollary 10.4.13 that each of them has an equivalent canonical representation of order at most t. Finally, it holds $\forall i \in [1,t] : |B_r| \leq 6$ and thus $|R| = \prod_{i=1}^t |B_i| \leq 6^t$. □

By applying the technique from the proof above to the prefix-palindromes, i.e., to each of the representations of order 1 produced by Corollary 10.4.3, we obtain the following result. (The difference with Corollary 10.4.3 is in the fact that here, the representations of the sets of ordrer 1 are canonical.)

Corollary 10.4.15. *The set of prefix-palindromes of a string $T[1..n]$ can be partitioned into $O(\log n)$ affine sets of order at most 1. Each set of order 1 has a* ***canonical*** *representation of the form $\langle U_i(V_iU_i)^{\ell_i}, (V_iU_i, 1, u_i) \rangle$ for some $U_i \in \mathsf{PAL} \cup \{\varepsilon\}$, $V_i \in \mathsf{PAL}$ and integers $\ell_i \geq 1$ and $u_i > 1$.*

Corollary 10.4.16. *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$ be a canonical representation of an affine prefix set. Then it holds $\forall r \in [1,t] : |Q_r| > \sum_{j=r+1}^t |Q_j^{u_j+4}|$.*

Proof. If $\rho = \langle X, (Q_i, \ell_i, u_i)_{i=1}^t \rangle$ is canonical, then clearly $\mathsf{expand}(\rho) = \langle X, (Q_i, \ell_i, u_i + 5)_{i=1}^t \rangle$ is irreducible. Thus, the statement follows from Lemma 10.4.6 applied to $\mathsf{expand}(\rho)$. □

Lemma 10.4.17. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t \rangle$ be a canonical representation of an affine prefix set, and let $h \in [0,5]$. Then $\langle \varepsilon, (Q_i, 1, u_i + h)_{i=2}^t \rangle$ is an irreducible representation of an affine prefix set of the string Q_1^2, and, if $h < 5$, also of the string Q_1.*

Proof. Consider any $h \in [0,5]$. Due to the strong affinity, $\langle X, (Q_i, 1, u_i + h)_{i=1}^t \rangle$ represents an affine prefix set. Let $(a_i)_{i=2}^t$ be a sequence of exponents with $\forall j \in [2,t] : a_j \in [1, u_i + h]$. By Lemma 10.4.5, the string $Q_1 Q_2^{a_2} Q_3^{a_3} \dots Q_t^{a_t}$ has period Q_1. Due to Lemma 10.4.6, it holds $|Q_2^{a_2} Q_3^{a_3} \dots Q_t^{a_t}| < |Q_1 Q_2| < |Q_1^2|$. Hence we have shown that $Q_2^{a_2} Q_3^{a_3} \dots Q_t^{a_t}$ is a prefix of Q_1^2, and $\langle \varepsilon, (Q_i, 1, u_i + h)_{i=2}^t \rangle$ is a representation of an affine prefix set of Q_1^2. Since $\langle X, (Q_i, 1, u_i)_{i=1}^t \rangle$ is irreducible, it is easy to see that also $\langle \varepsilon, (Q_i, 1, u_i + h)_{i=2}^t \rangle$ is irreducible.

If $h < 5$, then Lemma 10.4.6 invoked with $\rho = \langle X, (Q_i, 1, u_i + 5)_{i=1}^t \rangle$ implies that $|Q_2^{a_2} Q_3^{a_3} \dots Q_t^{a_t}| < |Q_1|$, and $\langle \varepsilon, (Q_i, 1, u_i + h)_{i=2}^t \rangle$ indeed only generates strings of length less than $|Q_1|$. □

Corollary 10.4.18. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t \rangle$ be a canonical representation of an affine prefix set. Then $\langle \varepsilon, (Q_i, 1, u_i)_{i=2}^t \rangle$ is a canonical representation of an affine prefix set of the string Q_1^2.*

Proof. By Lemma 10.4.17, $\langle \varepsilon, (Q_i, 1, u_i + 5)_{i=2}^t \rangle$ is an irreducible representation of an affine prefix set of Q_1^2. Hence $\langle \varepsilon, (Q_i, 1, u_i)_{i=2}^t \rangle$ is a canonical representation for Q_1^2. □

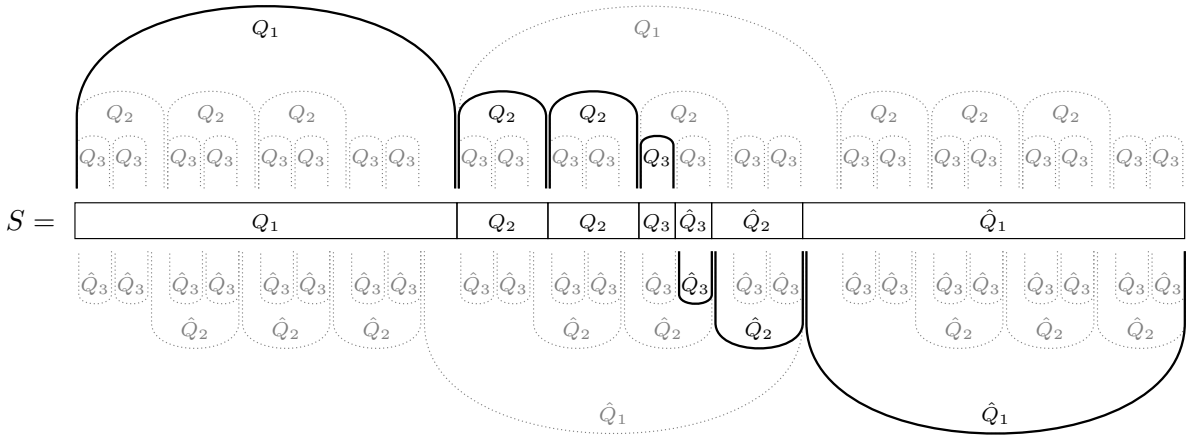

Figure 10.4: Lemma 10.4.19 applied to an irreducible representation $\langle X, (Q_1, 1, 2) \cdot (Q_2, 1, 3) \cdot (Q_3, 1, 2)\rangle$. The drawing shows the longest prefix $S = Q_1^2 Q_2^3 Q_3^2$ generated by the representation. By the lemma, for any $a_1 \in [0, 2]$, $a_2 \in [0, 3]$ and $a_3 \in [0, 2]$, it holds $S = Q_1^{2-a_1} Q_2^{3-a_2} Q_3^{2-a_3} \cdot \hat{Q}_3^{a_3} \hat{Q}_2^{a_2} \hat{Q}_1^{a_1}$, where each $\hat{Q}_j$ is the length-$|Q_j|$ suffix of S. The drawing highlights the case where $a_1 = a_2 = a_3 = 1$.

10.4.3 Reversing the structure of affine prefix sets

We first show that a periodic fragment of T induced by an affine prefix set can be covered by a combination of a forward and a "backward" affine prefix set. This is formally expressed by the lemma below, and visualized in Fig. 10.4.

Lemma 10.4.19. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be an irreducible representation of an affine prefix set, let $S = Q_1^{u_1} Q_2^{u_2} \dots Q_t^{u_t}$, and for $j \in [1, t]$ let $\hat{Q}_j$ be the length-$|Q_j|$ suffix of S. For any sequence $(a_i)_{i=1}^t$ with $\forall j \in [1, t] : a_j \in [0, u_j]$, it holds*

$$S = Q_1^{u_1-a_1} Q_2^{u_2-a_2} \dots Q_t^{u_t-a_t} \cdot \hat{Q}_t^{a_t} \hat{Q}_{t-1}^{a_{t-1}} \dots \hat{Q}_1^{a_1}.$$

Proof. If $t = 1$, then $S = Q_1^{u_1} = \hat{Q}_1^{u_1} = Q_1^{u_1-a_1} \hat{Q}_1^{a_1}$. Inductively assume that the lemma holds for representations of order $t - 1$. Now we show that it holds for representations of order t. If $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ is an irreducible representation of an affine prefix set, then clearly $\langle XQ_1^{u_1}, (Q_i, 1, u_i)_{i=2}^t\rangle$ is an irreducible representation of another affine prefix set. This representation is of order $t - 1$, and hence the inductive assumption implies

$$S = Q_1^{u_1} \cdot Q_2^{u_2-a_2} Q_3^{u_3-a_3} \dots Q_t^{u_t-a_t} \cdot \hat{Q}_t^{a_t} \hat{Q}_{t-1}^{a_{t-1}} \dots \hat{Q}_2^{a_2}.$$

If $a_1 = 0$, then there is nothing left to do. Hence assume $a_1 > 0$. Since $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ is an irreducible representation, Lemma 10.4.5 implies that $|Q_1|$ and therefore also $q = a_1 \cdot |Q_1|$ is a period of S. Hence S has a border of length $s - q$, where $s = |S|$, and it holds

$$S[1..s-q] = S[1+q..s] = Q_1^{u_1-a_1} \cdot Q_2^{u_2-a_2} Q_3^{u_3-a_3} \dots Q_t^{u_t-a_t} \cdot \hat{Q}_t^{a_t} \hat{Q}_{t-1}^{a_{t-1}} \dots \hat{Q}_2^{a_2}.$$

Finally, as mentioned before, $S[s-q+1..s]$ of length $q = a_1 \cdot |Q_1|$ has period $|Q_1|$. Hence $S[s-q+1..s] = (S[s-|Q_1|+1..s])^{a_1} = \hat{Q}_1^{a_1}$, which concludes the proof. □

We now build on this characterization to convert irreducible representations of affine prefix sets of S into irreducible representations of affine prefix sets of $\mathsf{rev}(S)$.

Corollary 10.4.20. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set, let $s = \sum_{i=2}^t (u_i+1)\cdot|Q_i|$, and for $j \in [1,t]$ let $\hat{Q}_j$ be the length-$|Q_j|$ suffix of $\mathsf{rot}^s(Q_1)$. Then $\langle \varepsilon, (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=2}^t\rangle$ represents an affine prefix set of $\mathsf{rev}(\mathsf{rot}^s(Q_1))$.*

Proof. Consider any sequence $(a_i)_{i=2}^t$ of exponents admitted by the representation, i.e., $\forall j \in [2,t] : a_j \in [1, u_j]$. By Lemma 10.4.17, $\langle \varepsilon, (Q_i, 1, u_i+1)_{i=2}^t\rangle$ is an irreducible representation of an affine prefix set of Q_1, which implies $Q_1[1..s] = Q_2^{u_2+1}Q_3^{u_3+1}\dots Q_t^{u_t+1}$. For this representation, Lemma 10.4.19 implies that $\hat{Q}_t^{a_t}\hat{Q}_{t-1}^{a_{t-1}}\dots\hat{Q}_2^{a_2}$ is a suffix of $Q_1[1..s]$. Thus, its reversal $\mathsf{rev}(\hat{Q}_t^{a_t}\hat{Q}_{t-1}^{a_{t-1}}\dots\hat{Q}_2^{a_2}) = \mathsf{rev}(\hat{Q}_2^{a_2})\mathsf{rev}(\hat{Q}_3^{a_3})\dots\mathsf{rev}(\hat{Q}_t^{a_t})$ is a prefix of $\mathsf{rev}(Q_1[1..s])$, which is a prefix of $\mathsf{rev}(\mathsf{rot}^s(Q_1))$. □

10.5 Appending a Palindrome to an Affine Prefix Set

In this section, we describe how to extend an affine prefix set $\mathcal{A}$ with a palindrome. This broadly means that we want to compute a union of multiple affine prefix sets, such that each of the new prefixes is the concatenation of a prefix in $\mathcal{A}$ and a palindrome. We distinguish two cases depending on whether or not the palindrome to be appended is inside a periodic fragment of T or not. Regardless of the case, we may first overextend $\mathcal{A}$ so that the new affine sets are not necessarily affine prefix sets. Whenever this happens, we truncate the sets by restricting the length of their strings with the auxiliary lemma below. For a set of strings $\mathcal{A}$, denote $\mathcal{A}|_m = \{S \in \mathcal{A} : |S| \le m\}$.

Lemma 10.5.1. *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be a representation of an affine prefix set $\mathcal{A}$. For $m \in \mathbb{N}$, we can express $\mathcal{A}' = \mathcal{A}|_m$ as a union of at most $t' \le t$ affine prefix sets $\mathcal{A}' = \bigcup_{j=1}^{t'}\mathcal{A}_j$, each with a representation of order at most t.*

Proof. Due to Lemma 10.4.10, we can assume that the given representation is irreducible. If $|XQ_1| > m$, then $\mathcal{A}'$ is empty, and thus it is the union of zero affine prefix sets. If $|XQ_1| \le m$ and $t = 1$, then we simply create a single new representation $\langle X, (Q_1, 1, \lfloor(m-|X|)/|Q_1|\rfloor)\rangle$ that represents $\mathcal{A}'$. The proof for $t > 1$ works by induction.

Assume that the lemma holds for representations of order at most $t-1$. Now we show that it holds for representations of order t. Let $a_1 \in \mathbb{N}$ be the minimal exponent such that $a_1 \ge \ell_1$ and $|XQ_1^{a_1}Q_2^{u_2}Q_3^{u_3}\dots Q_t^{u_t}| > m$. If $a_1 > u_1$, then $\mathcal{A} = \mathcal{A}'$ and there is nothing left to do. If $a_1 \le u_1$, then Lemma 10.4.6 (and the fact that the representation is irreducible) implies $|XQ_1^{a_1+1}Q_2| > |XQ_1^{a_1}Q_2^{u_2}Q_3^{u_3}\dots Q_t^{u_t}| > m$. Thus, we do not need to consider prefixes that are generated by using an exponent larger than a_1 for Q_1, and we partition the remaining prefixes into two affine prefix sets. The first one is $\mathcal{A}''$ represented by $\langle XQ_1^{a_1}, (Q_i, \ell_i, u_i)_{i=2}^t\rangle$, i.e., the set that contains all the strings that use exponent a_1 for Q_1. Its representation is of order $t-1$, and the inductive assumption implies that it is the union of $t'-1 \le t-1$ affine prefix sets $\mathcal{A}'' = \bigcup_{j=1}^{t'-1}\mathcal{A}_j$. For the remaining strings, if $a_1 > \ell_1$, we create one additional set $\mathcal{A}_{t'}$ represented by $\langle X, (Q_1, \ell_1, a_1-1)\cdot(Q_i, \ell_i, u_i)_{i=2}^t\rangle$. By minimality of a_1, all strings in $\mathcal{A}_{t'}$ have length at most m. It then holds $\mathcal{A}' = \mathcal{A}''\cup\mathcal{A}_{t'}$. If, however, $a_1 = \ell_1$, then it already holds $\mathcal{A}' = \mathcal{A}''$ and there is nothing left to do. □

10.5.1 Appending a long palindrome

Assume that the affine prefix set to be extended is given in a canonical representation $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$. We first focus on appending *long palindromes* of length at least $2|Q_1|$, and then we show that the shorter palindromes can be handled recursively. Note that,

for a canonical representation, T has a prefix $XQ_1^{u_1+5}$. At the same time, the longest prefix in the affine set is of length less than $XQ_1^{u_1+1}$. This leads us to a case distinction based on the center of the palindrome to be appended. If the center is before position $|XQ_1^{u_1+3}|$, then we can show that the entire palindrome is within the $|Q_1|$-periodic prefix of $T[|X|+1..n]$. Otherwise, the left half of the palindrome contains position $|XQ_1^{u_1+2}|$, and we can use this position as an anchor point for the extension.

10.5.1.1 Appending a long palindrome within a run of Q_1

We now focus on the case where the long palindrome to be appended is entirely within the $|Q_1|$-periodic prefix of $T[|X|+1..n]$. We proceed in two steps. First (in Theorem 10.5.2), we show how to append a palindrome under the assumption that the entire string has the form XQ_1^x for some integer x. The second step (Corollary 10.5.3) truncates the result of the first step such that it corresponds to XQ_1^α, where $\alpha \in \mathbb{Q}$ is the largest value such that XQ_1^α is a prefix of T.

Theorem 10.5.2. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$. Let $s = \sum_{i=2}^t (u_i+1)\cdot|Q_i|$, and for $j \in [1,t]$ let $\hat{Q}_j$ be the length-$|Q_j|$ suffix of $\mathsf{rot}^s(Q_1)$. If $\mathsf{rot}^r(Q_1) = \mathsf{rev}(Q_1)$ for some $r \in [s, s+|Q_1|)$, then*

$$\langle X\cdot Q_1\cdot Q_1[1..r-s], (\mathsf{rot}^{r-s}(Q_1), 1, x)\cdot(\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=2}^t\rangle \tag{10.2}$$

represents an affine prefix set $\mathcal{A}'$ of XQ_1^{x+3}, for any positive integer x. Additionally, each of the following holds:

1. *If $Y' \in \mathcal{A}'$, then there is a string $Y \in \mathcal{A}$ and a palindrome P such that $Y' = YP$.*
2. *For $Y \in \mathcal{A}$ and $P \in \mathsf{PAL}$, if $|P| \geq 2|Q_1|$ and YP is a prefix of XQ_1^{x+1}, then $YP \in \mathcal{A}'$.*

Proof. Let $q = |Q_1|$. By Corollary 10.4.16, it holds $s < q$. Consider any string $XS' \in \mathcal{A}'$. Since this string is generated by Eq. (10.2), there must be exponents $a_1 \in [1,x]$ and $\forall j \in [2,t] : a_j \in [1,u_j]$ such that

$$S' = Q_1\cdot Q_1[1..r-s]\cdot\mathsf{rot}^{r-s}(Q_1)^{a_1}\cdot Q' = Q_1^{a_1+1}\cdot Q_1[1..r-s]\cdot Q',$$

where $Q' = \mathsf{rev}(\hat{Q}_2)^{a_2}\mathsf{rev}(\hat{Q}_3)^{a_3}\dots\mathsf{rev}(\hat{Q}_t)^{a_t}$. We start by showing that Eq. (10.2) represents an affine prefix set of XQ_1^{x+3}, i.e., we must show that S' is a prefix of Q_1^{x+3}. The suffix Q' of S' was generated by the last part $(\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=2}^t$ of Eq. (10.2). Corollary 10.4.20 implies that Q' is a prefix of

$$\mathsf{rev}(\mathsf{rot}^s(Q_1)) = \mathsf{rot}^{-s}(\mathsf{rev}(Q_1)) = \mathsf{rot}^{-s}(\mathsf{rot}^r(Q_1)) = \mathsf{rot}^{r-s}(Q_1).$$

Therefore, it holds

$$S' = Q_1^{a_1+1}\cdot Q_1[1..r-s]\cdot(\mathsf{rot}^{r-s}(Q_1))[1..|Q'|] = Q_1^{a_1+1}\cdot(Q_1^2)[1..r-s+|Q'|],$$

and S' is a prefix of $Q_1^{a_1+3}$, which is a prefix of Q_1^{x+3}.

Next, we show that $S' = SP$ for some string S with $XS \in \mathcal{A}$ and a palindrome P. It holds $a_j \in [1,u_j]$ if and only if $u_j - a_j + 1 \in [1,u_j]$, and thus $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ generates the string $XS \in \mathcal{A}$ with $S = Q_1^1 Q_2^{u_2-a_2+1} Q_3^{u_3-a_3+1}\dots Q_t^{u_t-a_t+1}$, where $|S| = q+s-|Q'|$. Let P be the unique string such that $S' = SP$, i.e.,

$$P = S'[q+s-|Q'|+1..|S'|] = Q_1[s-|Q'|+1..q]\cdot Q_1^{a_1-1}\cdot(Q_1^2)[1..r-s+|Q'|].$$

It remains to be shown that P is a palindrome. Let $L = Q_1[s - |Q'| + 1..q]$ and $R = (Q_1^2)[1..r-s+|Q'|]$, then Corollary 10.2.4 implies that P is a palindrome if $\mathsf{rot}^{|R|-|L|}(Q_1) = \mathsf{rev}(Q_1)$. Indeed, cyclically shifting Q_1 by

$$|R| - |L| = (r - s + |Q'|) - (q - (s - |Q'| + 1) + 1) = r - q$$

steps is equivalent to cyclically shifting it by r steps, and thus $\mathsf{rot}^{|R|-|L|}(Q_1) = \mathsf{rot}^{r-q}(Q_1) = \mathsf{rot}^{r}(Q_1) = \mathsf{rev}(Q_1)$. Hence P is a palindrome.

Finally, consider any string $XS \in \mathcal{A}$ and a palindrome $P \in \mathsf{PAL}$ of length $|P| \geq 2|Q_1|$ such that SP is a prefix of Q_1^{x+1}. Since P is a substring of Q_1^{x+1}, it must be of the form LQ_1^zR for some positive integer z, a suffix L of Q_1, and a prefix R of Q_1. Due to $XS \in \mathcal{A}$, there is some sequence $\forall j \in [1,t] : a_j \in [1,u_j]$ of exponents such that $S = Q_1^{u_1-a_1+1}Q_2^{u_2-a_2+1}\dots Q_t^{u_t-a_t+1}$. Let $q' = \sum_{j=2}^{t} a_j \cdot |Q_j|$, then $L = Q_1[s - q' + 1..q]$ and $|R| = |P| - zq - q + s - q'$. Since P is a palindrome, Corollary 10.2.4 implies that $\mathsf{rot}^{|R|-|L|}(Q_1) = \mathsf{rev}(Q_1)$. Since Q_1 is primitive and $\mathsf{rev}(Q_1) = \mathsf{rot}^{r}(Q_1)$, it is necessary that $|R| - |L| = z'q + r$ for some integer z'. This leads to

$$|R| - |L| = |P| + 2(s - q') - zq - 2q = z'q + r,$$

or equivalently

$$|P| = (z' + z + 2)q + r - 2(s - q').$$

We have shown that SP is of length $|P| + |S| = (u_1 - a_1 + z' + z + 2)q + q + r - s + q'$ for some integers z, z' such that $|SP| \leq (x+1)q$. Let $x' = (u_1 - a_1 + z' + z + 2)$, and note that $|SP| \leq q(x+1)$ implies $x' \leq (q(x+1) - q - r + s - q')/q \leq x$. Finally, the representation stated in the lemma generates the string $XQ_1Q_1[1..r - s]\mathsf{rot}^{r-s}(Q_1)^{x'}\mathsf{rev}(\hat{Q}_2)^{a_2}\mathsf{rev}(\hat{Q}_3)^{a_3}\dots\mathsf{rev}(\hat{Q}_t)^{a_t}$ of length $|X| + q + r - s + qx' + q' = |XSP|$. Hence $XSP \in \mathcal{A}'$ as required. □

Corollary 10.5.3. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$. Let $\alpha \in \mathbb{Q}$ be the largest possibly fractional exponent such that XQ_1^α is a prefix of T, and define*

$$\mathcal{S} = \{S \cdot P : S \cdot P \text{ is a prefix of } XQ_1^\alpha, S \in \mathcal{A}, P \in \mathsf{PAL}, |P| \geq 2|Q_1|\}$$

There are $t' \leq t$ affine prefix sets $\mathcal{B}_i$, $i \in [1,t']$, each of order $\leq t$, such that both of the following properties hold for $\mathcal{B} = \bigcup_{i=1}^{t'} \mathcal{B}_i$:

1. *$\mathcal{S} \subseteq \mathcal{B}$.*
2. *For every $Y' \in \mathcal{B}$, there is a string $Y \in \mathcal{A}$ and a palindrome P such that $Y' = YP$.*

Proof. If $\mathcal{S}$ is empty, then $t' = 0$ trivially satisfies the claim of the lemma. Otherwise, note that any palindrome P considered by $\mathcal{S}$ is a substring of Q_1^α. Hence, if $\mathcal{S}$ is non-empty, there is a palindromic substring of Q_1^α that is of length at least $2|Q_1|$, and Corollary 10.2.4 implies that $\mathsf{rev}(Q_1)$ is a rotation of Q_1. Particularly, for arbitrary integer s, there is an integer $r \in [s, s + |Q_1|)$ such that $\mathsf{rev}(Q_1) = \mathsf{rot}^r(Q_1)$. This allows us to apply Theorem 10.5.2 to $XQ_1^{\lfloor\alpha\rfloor+3}$ to obtain an affine prefix set $\mathcal{A}'$ of order t satisfying each of the following:

1. If $Y' \in \mathcal{A}'$, then there is a string $Y \in \mathcal{A}$ and a palindrome P such that $Y' = YP$.
2. For $Y \in \mathcal{A}$ and $P \in \mathsf{PAL}$, if $|P| \geq 2|Q_1|$ and YP is a prefix of $XQ_1^{\lfloor\alpha\rfloor+1}$, then $YP \in \mathcal{A}'$.

Let $\mathcal{B} = \mathcal{A}'|_{|X|+\alpha\cdot|Q_1|}$. We have $\mathcal{S} \subseteq \mathcal{B}$ and for all $Y' \in \mathcal{B}$, there is a string $Y \in \mathcal{A}$ and a palindrome P such that $Y' = YP$. By Lemma 10.5.1, $\mathcal{B}$ is a union of $\leq t$ affine prefix sets of order $\leq t$. □

10.5.1.2 Appending a long palindrome outside a run of Q_1

Theorem 10.5.4. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$ and $s = \sum_{i=2}^t (u_i+1)\cdot|Q_i|$. For $j \in [1,t]$, let $\hat{Q}_j$ be the length-$|Q_j|$ suffix of $\mathsf{rot}^s(Q_1)$. For any string P,*

$$\langle X \cdot Q_1^{u_1+2} \cdot P \cdot \mathsf{rev}(Q_1)[1..|Q_1|-s], (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=1}^t\rangle$$

represents an affine prefix set $\mathcal{A}'$ of the string $X \cdot Q_1^{u_1+2} \cdot P \cdot \mathsf{rev}(Q_1^{u_1+2})$, where

$$\mathcal{A}' = \{SWP \cdot \mathsf{rev}(W) \mid S \in \mathcal{A} \text{ and } SW = X \cdot Q_1^{u_1+2}\}.$$

Proof. Let $q = |Q_1|$. We can split the output representation into a concatenation

$$\langle X \cdot Q_1^{u_1+2} \cdot P \cdot \mathsf{rev}(Q_1)[1..q-s], (\mathsf{rev}(\hat{Q}_1), 1, u_1)\rangle \cdot \langle \varepsilon, (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=2}^t\rangle. \tag{10.3}$$

By Corollary 10.4.20, $\langle \varepsilon, (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=2}^t\rangle$ represents an affine prefix set of $\mathsf{rev}(\hat{Q}_1)$. Hence any string generated by Eq. (10.3) is a prefix of $XQ_1^{u_1+2}P \cdot \mathsf{rev}(Q_1)[1..q-s] \cdot \mathsf{rev}(\hat{Q}_1)^{u_1+1}$. Due to $\mathsf{rev}(\hat{Q}_1) = \mathsf{rev}(\mathsf{rot}^s(Q_1)) = \mathsf{rev}(Q_1)[q-s+1..q] \cdot \mathsf{rev}(Q_1)[1..q-s]$, this string is in turn a prefix of $XQ_1^{u_1+2}P \cdot \mathsf{rev}(Q_1)^{u_1+2}$. We have shown that Eq. (10.3) represents an affine prefix set of the string $X \cdot Q_1^{u_1+2} \cdot P \cdot \mathsf{rev}(Q_1^{u_1+2})$.

Every element in $\mathcal{A}$ contributes exactly one element to $\mathcal{A}'$, and hence $|\mathcal{A}'| = |\mathcal{A}|$. Whether or not an affine representation is irreducible depends solely on the lengths and exponent bounds of its components. Since lengths and exponent bounds are identical for the two representations stated in the lemma, it is clear that both of them are irreducible. By Lemma 10.4.7, each of the two representations generates exactly $|\mathcal{A}| = |\mathcal{A}'| = \prod_{i=1}^t u_i$ distinct strings.

Since Eq. (10.3) generates exactly $|\mathcal{A}'|$ distinct strings, it suffices to show that any string generated by Eq. (10.3) is in $\mathcal{A}'$. It then readily follows that Eq. (10.3) generates exactly $\mathcal{A}'$. Thus, consider any string S' generated by Eq. (10.3). Such a string must be of the form $S' = XQ_1^{u_1+2}P \cdot \mathsf{rev}(W)$, where $\mathsf{rev}(W) = \mathsf{rev}(Q_1)[1..q-s] \cdot \mathsf{rev}(\hat{Q}_1)^{a_1}\mathsf{rev}(\hat{Q}_2)^{a_2}\ldots\mathsf{rev}(\hat{Q}_t)^{a_t}$ for some exponents $\forall i \in [1,t] : a_1 \in [1,u_i]$. By our previous observations, $\mathsf{rev}(W)$ is a prefix of $\mathsf{rev}(Q_1)^{u_1+2}$, and thus there is a unique string S such that $SW = XQ_1^{u_1+2}$ and $S' = SWP \cdot \mathsf{rev}(W)$. It remains to be shown that $S \in \mathcal{A}$, which then implies $S' \in \mathcal{A}'$. For this purpose, we carefully analyze the length of S.

$$\begin{aligned} |S| \;=\; |XQ_1^{u_1+2}| - |W| \;&=\; |XQ_1^{u_1+2}| - (q-s) - \sum\nolimits_{i=1}^t a_i \cdot |Q_i| \\ &=\; |XQ_1^{u_1-a_1+1}| + s - \sum\nolimits_{i=2}^t a_i \cdot |Q_i| \\ &=\; |X| + \sum\nolimits_{i=1}^t (u_i - a_i + 1) \cdot |Q_i| \end{aligned}$$

Recall that $\forall i \in [1,t] : a_i \in [1,u_i]$ and thus $(u_i - a_i + 1) \in [1,u_1]$. This means that $S = XQ_1^{u_1-a_1+1}Q_2^{u_2-a_2+1}\ldots Q_t^{u_t-a_t+1}$ is indeed in $\mathcal{A}$, which concludes the proof. □

If a fragment $P = T[x..y]$ of T is a palindrome, denote its center $(x+y)/2$ by $\mathrm{cen}(P)$.

Corollary 10.5.5. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$, and consider the set of strings*

$$\mathcal{A}' = \{S \cdot P : S \cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}, \mathrm{cen}(P) > |XQ_1^{u_1+3}|\}.$$

There are $t' = O(t \log n)$ affine prefix sets $\mathcal{B}_i$, $i \in [1,t']$, each of order $\le t+1$, such that both of the following properties hold for $\mathcal{B} = \bigcup_{i=1}^{t'} \mathcal{B}_i$:

1. $\mathcal{A}' \subseteq \mathcal{B}$.
2. *For every* $Y' \in \mathcal{B}$, *there is a string* $Y \in \mathcal{A}$ *and a palindrome* P *such that* $Y' = YP$.

Proof. Consider any $SP \in \mathcal{A}'$, where SP is a prefix of T, $S \in \mathcal{A}$, $P \in \mathsf{PAL}$, $\mathrm{cen}(P) > |XQ_1^{u_1+3}|$. Due to $S \in \mathcal{A}$, Corollary 10.4.16 implies $|S| < |XQ_1^{u_1+1}|$. Let $P' = T[x..y]$, where $x = 1 + |XQ_1^{u_1+2}|$ and $y = 2 \cdot \mathrm{cen}(P) - x$. We claim that $P' \in \mathsf{PAL}$. Indeed, the starting position $|S| + 1$ of P is less than the starting position x of P', and the centers of P and P' coincide with $\mathrm{cen}(P) - x = y - \mathrm{cen}(P)$. We call P' the *core palindrome* of SP. Note that every core palindrome is a prefix of $T[x..n]$ (which is independent of SP). Therefore, by Corollary 10.4.15, the set of core palindromes can be represented as the union of $O(\log n)$ affine prefix sets. Let $\mathcal{C}$ be any of these sets. We now describe how to compute the part of $\mathcal{A}'$ that contains strings of the form $SP = SWP' \cdot \mathsf{rev}(W)$, where $S \in \mathcal{A}$, $P \in \mathsf{PAL}$, and the core palindrome of SP is some $P' \in \mathcal{C}$. The procedure depends on the representation of $\mathcal{C}$, which, by Corollary 10.4.15, is covered by one of the following cases. Let $q = |Q_1|$.

Case 1: $\mathcal{C}$ is given in strongly affine representation $\langle U \cdot (VU)^\ell, (VU, 1, u)\rangle$, where VU is primitive and $|VU| > q$. For any $i \in [1, u]$, let $P_i = U \cdot (VU)^{\ell+i}$ be a core palindrome in $\mathcal{C}$. Using Theorem 10.5.4, we compute an affine prefix set $\mathcal{C}_i = \{SWP_i \cdot \mathsf{rev}(W) \mid S \in \mathcal{A} \text{ and } SW = X \cdot Q_1^{u_1+2}\}$ of the string $XQ_1^{u_1+2}P_i \cdot \mathsf{rev}(Q_1)^{u_1+2}$ (not necessarily a prefix of T) with a representation

$$\langle X \cdot Q_1^{u_1+2} \cdot P_i \cdot \mathsf{rev}(Q_1)[1..|Q_1| - s], (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=1}^t\rangle.$$

If $\mathcal{A}'$ contains some string $SWP_i \cdot \mathsf{rev}(W) = XQ_1^{u_1+2}P_i \cdot \mathsf{rev}(W)$, then this string is clearly also in $\mathcal{C}_i$. Also, every string in $\mathcal{C}_i$ is the concatenation of some string in $\mathcal{A}$ and a palindrome. However, there may be some strings in $\mathcal{C}_i$ that are not prefixes of T if $XQ_1^{u_1+2}P_i \cdot \mathsf{rev}(Q_1)^{u_1+2}$ is not a prefix of T. We have already established that $XQ_1^{u_1+2}P_i$ is a prefix of T, and we split the representation of $\mathcal{C}_i$ into two parts by defining a set $\mathcal{W}_i$ such that $\mathcal{C}_i = \{XQ_1^{u_1+2}P_i \cdot \mathsf{rev}(W) \mid \mathsf{rev}(W) \in \mathcal{W}_i\}$. The first part has representation $\langle XQ_1^{u_1+2}P_i, \varepsilon\rangle$ of order 0. The second part $\mathcal{W}_i$ has representation

$$\langle \mathsf{rev}(Q_1)[1..|Q_1| - s], (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=1}^t\rangle$$

and contains prefixes of $\mathsf{rev}(Q_1)^{u_1+2}$. Let $x = 1 + |XQ_1^{u_1+2}|$ and $y_i = x + |P_i| - 1$, i.e., $T[x..y_i] = P_i$. Our goal is to truncate $\mathcal{W}_i$ such that we remove exactly all the strings that are not prefixes of $T[y_i + 1..n]$. Note that all the $\mathcal{W}_i$'s are identical, i.e., they are independent of i, and we will show that they remain identical even after truncating.

Recall that the given representation of $\mathcal{C}$ is strongly affine, and hence, $T[x..n]$ has a prefix $U \cdot (VU)^{\ell+u+5} = P_i \cdot (VU)^{5+u-i}$ (for any $i \in [1, u]$). This implies that $T[y_i + 1..n]$ has a prefix $(VU)^2$. Let $\alpha \in \mathbb{Q}$ be the largest (possibly fractional exponent) such that $\mathsf{rev}(Q_1)^\alpha$ is a prefix of $(VU)^2$. Since (VU) is primitive, and due to $q < |VU|$, it cannot be that q is a period of $(VU)^2$ (see Lemma 10.2.1). Hence $|\mathsf{rev}(Q_1)^\alpha| < |(VU)^2|$, and α is the maximal exponent such that $\mathsf{rev}(Q_1)^\alpha$ is a prefix of $T[y_i + 1..n]$. We apply Lemma 10.5.1 and obtain $\mathcal{W}_i|_{(\alpha \cdot q)}$ as the union of $t' \le t$ representations, each of order at most t. Note that α is independent of i, and thus all the $\mathcal{W}_i|_{(\alpha \cdot q)}$ are indeed identical.

Finally, let $\mathcal{W}' = \mathcal{W}_1|_{(\alpha \cdot q)}$. We must represent the union of all the truncated $\mathcal{C}_i$'s, defined by $\mathcal{B}(\mathcal{C}) = \{XQ_1^{u_1+2} \cdot P_i \cdot \mathsf{rev}(W) \mid i \in [1, u], \mathsf{rev}(W) \in \mathcal{W}'\}$. This set can be readily obtained by creating t' copies of $\langle XQ_1^{u_1+2}U(VU)^\ell, (VU, 1, u)\rangle$, and concatenating each copy with one of the t' representations of order at most t that make up $\mathcal{W}'$. Then, $\mathcal{B}(\mathcal{C})$ is the union of t affine prefix sets, each of order at most $t + 1$.

Case 2: $\mathcal{C}$ is given in representation $\langle P', \varepsilon\rangle$ of order 0, i.e., it contains a single core palindrome P'. We proceed exactly like in Case 1, but with a single palindrome $P_1 = P'$. In Case 1, we only use the periodic structure of the P_i's to show that all the truncated sets $\mathcal{W}_i|_{(\alpha\cdot q)}$ are identical. Since this time we are only concerned with a single core palindrome, we simply let $\alpha \in \mathbb{Q}$ be the maximal value such that $\mathsf{rev}(Q_1)^\alpha$ is a prefix of $T[y_1+1..n]$. We then continue just like in Case 1, performing the final step with the fixed set $\langle XQ_1^{u_1+2}P_1, \varepsilon\rangle$ instead of $\langle XQ_1^{u_1+2}U(VU)^\ell, (VU, 1, u)\rangle$. This results in a set $\mathcal{B}(\mathcal{C})$ that is the union of at most t representations, each of order at most t.

Case 3: $\mathcal{C}$ has strongly affine representation $\langle U\cdot(VU)^\ell, (VU, 1, u)\rangle$, where VU is primitive and $|VU| = q$. For $i \in [1, u]$, let $P_i = U\cdot(VU)^{\ell+i}$. We show that, if $\mathcal{A}'$ contains some $SP = S\cdot W\cdot P_i\cdot\mathsf{rev}(W) = XQ_1^{u_1+2}P_i\cdot\mathsf{rev}(W)$ with $S \in \mathcal{A}$, then the entire SP can be written as XQ_1^α for some $\alpha \in \mathbb{Q}$. By the definition of a core palindrome, the center of P_i is $\mathrm{cen}(P_i) > |XQ_1^{u_1+3}|$, and its length is $2\cdot(\mathrm{cen}(P_i) - x) + 1 \geq 2q$. The entire palindrome $P = WP_i\cdot\mathsf{rev}(W)$ is then also of length at least $2q$. It holds $P_i[1..2q] = Q_1^2$ because T has a prefix $XQ_1^{u_1+5}$ (since $\mathcal{A}$ is given in strongly affine representation) and P_i starts at position $x = 1 + |XQ_1^{u_1+2}|$. It follows $SWP_i[1..2q] = XQ_1^{u_1+4}$ with $|S| > |X|$. Therefore, P has the q-periodic prefix WP_i of length $(|P| + |P_i|)/2 \geq |P|/2 + q > 3q/2$, and Lemma 10.2.2 implies that P has period q. We have established that $SWP_i[1..q] = XQ_1^{u_1+3}$, and that $P_i\cdot\mathsf{rev}(W)$ has period q. Since these fragments overlap by q letters, it is clear that $SWP_i\cdot\mathsf{rev}(W)$ is of the form XQ_1^α for some exponent $\alpha \in \mathbb{Q}$.

We have shown that Case 3 is only concerned with prefixes of the form XQ_1^α and with palindromes of length at least $2q$. Hence we can simply apply Corollary 10.5.3 and obtain a set $\mathcal{B}(\mathcal{C})$ as the union of at most t representations of order at most t. This set then contains every $SP = SWP_i\cdot\mathsf{rev}(W) = XQ_1^{u_1+2}P_i\cdot\mathsf{rev}(W) \in \mathcal{A}'$ for all the core palindrome sets $\mathcal{C}$ that fall into Case 3, and Corollary 10.5.3 guarantees that any element in $\mathcal{B}(\mathcal{C})$ is indeed the concatenation of a string in $\mathcal{A}$ and a palindrome.

Case 4: $\mathcal{C}$ has strongly affine representation $\langle U\cdot(VU)^\ell, (VU, 1, u)\rangle$, where VU is primitive and $|VU| < q$. As before, let $x = 1 + |XQ_1^{u_1+2}|$, and let $P_i = U\cdot(VU)^{\ell+i}$. As seen in Case 3, P_i is of length at least $2q$ and has prefix $P_i[1..2q] = Q_1^2$. Since P_i has a period $|VU| < q$, its prefix Q_1^2 also has a period $|VU|$. However, this contradicts the fact that Q_1 is primitive (see Lemma 10.2.1).

The list of cases is clearly exhaustive, and it remains to analyze the number of created representations. There are $O(\log n)$ core palindrome sets, and each set is covered by exactly one case. In every case, we create at most t representations, each of order at most $t + 1$, matching the statement of the corollary. □

10.5.1.3 Appending all long palindromes

Lemma 10.5.6. *Let $\langle X, (Q_i, 1, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$. Define the set of strings*

$$\mathcal{A}' = \{S\cdot P \mid S \in \mathcal{A}, P \in \mathsf{PAL}, \text{ and } S\cdot P \text{ is a prefix of } T\}.$$

There are $t' = O(t\log n)$ affine prefix sets $\mathcal{B}_i$, $1 \leq i \leq t'$, each of order at most $t+1$, that satisfy both of the following:

1. $\mathcal{A}' \subseteq \cup_{i=1}^{t'}\mathcal{B}_i$.

2. *For each string* $S' \in \cup_{i=1}^{t'} \mathcal{B}_i$, *there is a string* $S \in \mathcal{A}$ *and* $P \in \mathsf{PAL}$ *such that* $S' = S \cdot P$.

Proof. We consider the sets from Corollary 10.5.3 and Corollary 10.5.5, defined by

$$\mathcal{A}_1 = \{S \cdot P : S \cdot P \text{ is a prefix of } XQ_1^\alpha, S \in \mathcal{A}, P \in \mathsf{PAL}, |P| \geq 2|Q_1|\} \text{ and}$$

$$\mathcal{A}_2 = \{S \cdot P : S \cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}, \mathrm{cen}(P) > |X| + (u_1 + 3) \cdot |Q_1|\},$$

where α is the largest (possibly fractional) exponent such that XQ_1^α is a prefix of T. Due to Corollary 10.5.3 and Corollary 10.5.5, we can express (a superset of) $\mathcal{A}_1 \cup \mathcal{A}_2$ as the union of at most $O(t \log n)$ affine prefix sets, each of order at most $t + 1$, where every string in each of the prefix sets is the concatenation of a string from $\mathcal{A}$ and a palindrome.

It remains to be shown that $\mathcal{A}' \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$. For the sake of contradiction, assume that there is some string $SP \in \mathcal{A}' \setminus (\mathcal{A}_1 \cup \mathcal{A}_2)$, where $S \in \mathcal{A}$, $P \in \mathsf{PAL}$ and $|P| \geq 2|Q_1|$. Due to $SP \notin \mathcal{A}_1$, it cannot be that SP is a prefix of XQ_1^α. Thus, SP must be longer than XQ_1^α. Let $m = |XQ_1^\alpha| - |S|$, then the length-m suffix of Q_1^α is a prefix of P. We show a lower bound on m. Since the given representation is strongly affine, it holds $\alpha \geq u_1 + 5$. It is also irreducible, and hence Corollary 10.4.16 implies $|S| < |XQ_1^{u_1+1}|$. Therefore, it holds $m > 4|Q_1|$. Note that P does not have period $|Q_1|$, but its length-m prefix does. Hence, by Lemma 10.2.2, it follows that P is of length over $2m - |Q_1|$, and therefore

$$\mathrm{cen}(P) \geq |S| + |P|/2 > |S| + m - |Q_1|/2 = |XQ_1^\alpha| - |Q_1|/2 > |XQ_1^{u_1+4}|.$$

This implies $SP \in \mathcal{A}_2$, which contradicts the initial assumption. □

10.5.2 Recursively appending shorter palindromes and the final result

We have shown that appending palindromes of length at least $2|Q_1|$ results in at most $O(t \log n)$ affine prefix sets of order at most $t + 1$. For appending shorter palindromes, we will exploit properties of strongly affine prefix sets that allow us to apply the previously described approach recursively.

Lemma 10.5.7. *Let* $\langle X, (Q_i, 1, u_i)_{i=1}^t \rangle$ *be a canonical representation of an affine prefix set* $\mathcal{A}$. *Define the set of strings*

$$\mathcal{A}' = \{S \cdot P : S \cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}\}.$$

There are $t' = O((t + 1)^2 \log n)$ *affine prefix sets* $\mathcal{B}_i$, $1 \leq i \leq t'$, *each of order at most* $t + 1$, *such that* $\mathcal{A}' = \cup_{i=1}^{t'} \mathcal{B}_i$.

Proof. For the sake of induction, consider the case where $t = 0$. It holds $\mathcal{A} = \{X\}$ and $\mathcal{A}' = \{XP \mid P \in \mathsf{PAL} \text{ and } P \text{ is a prefix of } T[1 + |X| \,..n]\}$. By Corollary 10.4.15, we can express the prefix-palindromes of $T[1 + |X| \,..n]$ as the union of $O(\log n)$ affine prefix sets, and by prepending X we immediately obtain the statement of the lemma.

Now we show that the lemma holds for representations of order $t > 0$, inductively assuming that we have already shown the correctness for representations of order $t - 1$. We apply Lemma 10.5.6 and obtain $O(t \log n)$ affine prefix sets, each of order at most $t + 1$. These sets correspond to a superset of the prefixes in

$$\mathcal{A}'_{\text{long}} = \{S \cdot P : S \cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}, |P| \geq 2|Q_1|\},$$

and each prefix contained in any of the sets is the concatenation of an element in $\mathcal{A}$ with a palindrome. It remains to be shown how to cover

$$\mathcal{A}'_{\text{short}} = \{S \cdot P : S \cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}, |P| < 2|Q_1|\}.$$

By Corollary 10.4.18, $\langle \varepsilon, (Q_i, 1, u_i)_{i=2}^{t} \rangle$ is a canonical representation of an affine prefix set $\mathcal{A}''$ of Q_1^2. By the inductive assumption, the set

$$\mathcal{A}''' = \{S \cdot P : S \cdot P \text{ is a prefix of } Q_1^3, S \in \mathcal{A}'', P \in \mathsf{PAL}\}$$

is the union of $t'' = O(t^2 \log n)$ affine prefix sets $\mathcal{B}_j$, $j \in [1, t'']$ of order at most t. For any set $\mathcal{B}_j$, let $\langle X', (Q'_i, \ell'_i, u'_i)_{i=1}^{t_j} \rangle$ be a representation of order $t_j \leq t$. We obtain the set $\mathcal{B}'_j$ defined by the representation $\langle X, (Q_1, 1, u_1) \cdot (Q'_i, \ell'_i, u'_i)_{i=1}^{t_j} \rangle$ of order $t_j + 1 \leq t + 1$. To conclude the proof we show the following claims:

1. Each set $\mathcal{B}'_j$ is an affine prefix set of T.
2. Every element in any of the sets $\mathcal{B}'_j$ is also in $\mathcal{A}'$.
3. Every element in $\mathcal{A}'_{\text{short}}$ is contained in at least one set $\mathcal{B}'_j$.

For Claim 1, observe that a string in $\mathcal{B}_j$ is a prefix of Q_1^3, and thus a string in $\mathcal{B}'_j$ is a prefix of $XQ_1^{u_1+3}$. Since the original representation is strongly affine, $XQ_1^{u_1+3}$ is a prefix of T, and the correctness of the claim follows.

For Claim 2, every element in $\mathcal{B}'_j$ is of the form $XQ_1^{a_1}SP$ for a palindrome P, some $S \in \mathcal{A}''$, and $a_1 \in [1, u_1]$. Note that elements in $\mathcal{A}''$, hence also S, are of the form $Q_2^{a_2}Q_3^{a_3}\dots Q_t^{a_t}$ with $\forall i \in [2, t] : a_i \in [1, u_i]$. Thus, $XQ_1^{a_1}S$ is in $\mathcal{A}$, and $XQ_1^{a_1}SP$ is in $\mathcal{A}'$ as claimed.

For Claim 3, consider an element $SP \in \mathcal{A}'_{\text{short}}$ where $S \in \mathcal{A}$ and $P \in \mathsf{PAL}$ with $|P| < 2|Q_1|$. We can write S as $XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t}$ with $\forall i \in [1, t] : a_i \in [1, u_i]$. By Corollary 10.4.16, $Q_2^{a_2}\dots Q_t^{a_t}$ is of length less than Q_1, and SP is of length less than $|XQ_1^{a_1+3}| \leq |XQ_1^{u_1+3}|$. Since the original representation is strongly affine, $XQ_1^{u_1+3}$ is a prefix of T. Since also $XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t}P$ is a prefix of T, it is easy to see that $Q_2^{a_2}\dots Q_t^{a_t}P$ is a prefix of Q_1^3. Finally, $Q_2^{a_2}\dots Q_t^{a_t}$ is in $\mathcal{A}''$, and thus $Q_2^{a_2}\dots Q_t^{a_t}P$ is in $\mathcal{A}'''$. It follows that $SP = XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t}P$ is in one of the $\mathcal{B}'_j$.

We created $O(t \log n)$ representations of order at most $t + 1$ that cover $\mathcal{A}'_{\text{long}}$, and $t'' = O(t^2 \log n)$ representations of order at most $t+1$ that cover $\mathcal{A}'_{\text{short}}$. Hence we created $O((t + 1)^2 \log n)$ representations of order at most $t + 1$, as required by the lemma. □

Theorem 10.1.1. *Let $0 < \epsilon < 1$ be constant. Let T be a string of length n and let $k \in \mathbb{N}^+$. The set of prefixes of T that belong to PAL^k is the union of $O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$ affine prefix sets, each of order at most k.*

Proof. We start with the empty affine prefix set representing PAL^0. We proceed in k levels $k' \in [0, k)$, and, on each level k', we consider affine prefix sets of order k'. The union of all the affine prefix sets of level k' are exactly all of the prefixes of T that are in $\mathsf{PAL}^{k'}$. For each affine prefix set of the current level k', we first apply Lemma 10.4.10 and Lemma 10.4.14 to obtain $6^{k'}$ canonical representations of order at most k'. Then, for each of the representations, we append a palindrome using Lemma 10.5.7, resulting in $c \cdot (k' + 1)^2 \log n$ affine prefix sets of order at most $k' + 1$, which we move to level $k' + 1$. Here, c is a positive constant that depends on the precise complexity analysis of Lemma 10.5.7. Hence, after processing level $k - 1$, the total number of affine prefix sets is bounded by

$$\prod_{k'=0}^{k-1} (6^{k'} \cdot c \cdot (k' + 1)^2 \log n) \leq (k!)^2 \cdot c^k \cdot 6^{(k^2/2)} \cdot \log^k n.$$

Let $\epsilon \in \mathbb{R}^+$ with $\epsilon < 1$ be constant. If k exceeds another constant that depends solely on ϵ and c, then $(k!)^2 \cdot c^k < 6^{\epsilon' \cdot k^2}$ with $\epsilon' = \frac{1}{2-\epsilon} - \frac{1}{2} > 0$. Hence the bound becomes $6^{\epsilon' \cdot k^2} \cdot 6^{(k^2/2)} \cdot \log^k n = 6^{k^2/(2-\epsilon)}$. □

10.6 Read-only Algorithm for Encoding *k*-palindromic Prefixes and Computing the Palindromic Length

Finally, using the combinatorial properties we showed above, we develop a read-only algorithm that receives a string T of length n and an integer k, and computes a small space-representation of i-palindromic prefixes of a string for all $1 \leq i \leq k$. Additionally, it can compute the palindromic length of T if its at most k.

The following lemma gives a small-space implementation of the procedure behind Corollary 10.4.3 to compute the prefix-palindromes of a string as $O(\log n)$ affine prefix sets of order at most 1. The general idea is to enumerate the prefix-palindromes in the increasing order of length, using constant-space exact pattern matching.

Lemma 10.6.1. *There is a read-only algorithm that enumerates all prefix-palindromes of a string $T[1..n]$ in the increasing order of length in $O(n)$ time and $O(1)$ space.*

Proof. For a fixed $j \in [0, \lceil \log_2 n \rceil]$, we show how to enumerate the prefix-palindromes of length within range $[2^j, 2^{j+1})$. Consider any length $m \in [2^j, 2^{j+1})$. It is easy to see that $T[1..m]$ is a palindrome if and only if $\mathsf{rev}(T[1..2^j]) = T(m-2^j..m]$ because $T[1..2^j]$ spans more than half of $T[1..m]$. Recall that for strings X, Y, a fragment $Y[i..j] = X$ is an occurrence of X in Y. We use constant space and linear time pattern matching (see [78] and references therein) to enumerate all occurrences of a string $\mathsf{rev}(T[1..2^j])$ in $T[1..2^{j+1})$ in the left-to-right order. Whenever the pattern matching algorithm outputs an occurrence $T[i..i+2^j) = \mathsf{rev}(T[1..2^j])$, we output that $T[1..m]$ with $m = i + 2^j - 1$ as a palindromic prefix. By the previous observation, this reports all palindromic prefixes of length within range $[2^j, 2^{j+1})$ in the increasing order of length, using constant space and $O(2^j)$ time. Hence the total time for all $j \in [0, \lceil \log_2 n \rceil]$ is $O(\sum_{j=1}^{\lceil \log_2 n \rceil} 2^j) = O(n)$. □

Algorithm 10.6.2 (Implementation of Corollary 10.4.15)**.** *Given a string $T[1..n]$, there is a read-only algorithm that uses $O(\log n)$ space and $O(n)$ time and outputs all prefixes of T that belong to* PAL *as $O(\log n)$ affine sets of order at most 1. Each set of order 1 is reported in canonical representation $\langle U(VU)^\ell, (VU, 1, u)\rangle$ for some $U \in \mathsf{PAL} \cup \{\varepsilon\}$, $V \in \mathsf{PAL}$ and integers $\ell \geq 1$ and $u > 1$.*

Proof. We use the procedure described in the proof of Corollary 10.4.3. We start with a single prefix-palindrome represented by $\langle T[1], \varepsilon\rangle$. Then, we use Lemma 10.6.1 to enumerate the remaining prefix-palindromes in increasing order of length. Let P' be the most recently reported prefix-palindrome (initially $P' = T[1]$). Whenever some prefix $P = T[1..m]$ is reported to be a palindrome, we consider two cases based on whether or not $|P| > 3|P'|/2$. We proceed as in the proof of Corollary 10.4.3, storing all affine sets computed during the procedure (where creating or modifying a set takes constant time, performing simple arithmetic operations that depend solely on $|P|$ and $|P'|$). The representations may not be strongly affine, but the postprocessing described in Lemma 10.4.14 (and used to obtain Corollary 10.4.15) can be easily performed in constant time for each affine prefix set. (This holds because the present representations are of order 1.) There are at most $O(\log n)$ affine sets, hence the algorithm uses $O(\log n)$ space and $O(n)$ time. □

Theorem 10.1.3. *Let $0 < \epsilon < 1$ be constant. Given a string T of length n and $k \in \mathbb{N}^+$, there is a read-only algorithm that returns a compressed representation of all prefixes of T that belong to PAL^i, for each $i \in [1, k]$, in $O(n \cdot 6^{k^2/(2-\epsilon)} \cdot \log^k n)$ time and $O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$ space.*

Proof. We first provide algorithmic implementations of the combinatorial lemmas we showed above (rather straightforward, but we still provide them for completeness), and then combine them into the final algorithm.

We assume that a representation $\langle X, (Q_i, \ell_i, u_i)\rangle$ of an affine set is stored as a list of the components Q_i, associated with ℓ_i, u_i, and X is stored separately.

Transforming representations

Algorithm 10.6.3 (Implementation of Lemma 10.4.8)**.** *Given a pointer to the i-th component of the representation of an affine set, the four operations* switch_i, merge_i, split_i, *and* $\mathsf{truncate}$ *can each be performed in constant time and space.*

These four operations only change ℓ_i, u_i, or replace Q_i with its rotation, which can be computed via a constant number of arithmetic operations. In case of merge_i, the $(i+1)$-th element in the list of components needs to be deleted, while split_i requires an insertion between the i-th and $(i+1)$-th element. For switch_i, we need to swap the i-th and $(i+1)$-th element. Given a pointer to the i-th element, the required deletions, insertions, and swaps take constant time. Hence all four operations can be implemented in $O(1)$ time and $O(1)$ extra space. As a corollary, we obtain the following:

Algorithm 10.6.4 (Implementation of Lemma 10.4.9)**.** *Given the representation of an affine prefix set of order t, we can remove all fixed components in $O(t^2)$ time and $O(1)$ space.*

Algorithm 10.6.5 (Implementation of Lemma 10.4.10)**.** *Let $\mathcal{A}$ be an affine prefix set of $T[1..n]$, given in representation ρ of order t. There is a read-only algorithm that, given ρ and random access to T, transforms ρ into an irreducible representation ρ^* of $\mathcal{A}$ in $O(t^2)$ time and $O(1)$ additional space. The representation ρ^* is of order at most t.*

We assume to receive an affine prefix set with a representation of order t, and transform it into an irreducible representation. The transformation starts with an application of Lemma 10.4.9, which requires $O(t^2)$ time and $O(1)$ extra space. It then performs $\leq t$ merge operations and $\leq t$ split operations in $O(t^2)$ total time and $O(1)$ extra space. Finally, the transformation applies Lemma 10.4.9 again in $O(t^2)$ time and $O(1)$ extra space.

Strongly affine representations

Algorithm 10.6.6 (Implementation of Lemma 10.4.14)**.** *Given an affine prefix set $\mathcal{A}$ as a representation of order t, there is a read-only algorithm that computes, in $O(6^t \cdot t^2)$ total time and $O(6^t \cdot t)$ space, at most 6^t canonical representations of order at most t of affine sets that form a partition of $\mathcal{A}$.*

We start by applying Algorithm 10.6.5 to make the representation irreducible in $O(t^2)$ time and $O(1)$ space. We then create 6^t representations of Eq. (10.1) naively in $O(6^t)$ time and $O(6^t \cdot t)$ space, and finally once more apply Algorithm 10.6.5 to each representation in total $O(6^t \cdot t^2)$ time and $O(6^t \cdot t)$ space. As observed in Lemma 10.4.14, this indeed results in canonical representations.

Appending a new palindrome

Algorithm 10.6.7 (Implementation of Lemma 10.5.1)**.** *Given an affine prefix set $\mathcal{A}$ with its representation $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ and an integer $m \in \mathbb{N}$, there is a read-only algorithm that finds a decomposition of $\mathcal{A}|_m$ into at most t affine prefix sets, each of order at most t, in $O(t^2)$ time and space.*

The representations of the sets are computed in a loop over $i = 1, 2, \ldots, t$. For a fixed i, the task is, given exponents $a_1, a_2, \ldots, a_{i-1}$, to compute the minimal $a_i \in [\ell_i, u_i]$ such that

$$|XQ_1^{a_1}Q_2^{a_2}\ldots Q_{i-1}^{a_{i-1}}Q_i^{a_i}Q_{i+1}^{u_{i+1}}\ldots Q_t^{u_t}| \geq m.$$

We then add a set with a representation $\langle XQ_1^{a_1}Q_2^{a_2}\ldots Q_{i-1}^{a_{i-1}}, (Q_i, \ell_i, a_i{-}1)\cdot(Q_j, \ell_j, u_j)_{j=i+1}^t\rangle$ to the union, but only if $a_i > \ell_i$. Either way, we continue with $i+1$. We can obtain a_i by first computing $m' = |XQ_1^{a_1}Q_2^{a_2}\ldots Q_{i-1}^{a_{i-1}}Q_{i+1}^{u_{i+1}}\ldots Q_t^{u_t}|$, which takes $O(t)$ time. Then, it holds that $a_i = \lceil (m-m')/|Q_i|\rceil$ (unless this value is not in $[\ell_i, u_i]$, which can be handled trivially). Creating the new representation takes $O(t)$ time and space, and the overall time and space for all i is $O(t^2)$.

Algorithm 10.6.8 (Implementation of Corollary 10.5.3)**.** *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$. Let α be the largest (possibly fractional) exponent such that XQ_1^α is a prefix of T and define*

$$\mathcal{S} = \{S\cdot P : S\cdot P \text{ is a prefix of } XQ_1^\alpha, S \in \mathcal{A}, P \in \mathsf{PAL}, |P| \geq 2|Q_1|\}$$

There is a read-only algorithm that computes in $O(n)$ time and $O(t^2)$ space at most t affine prefix sets $\mathcal{B}_i$ such that their union covers $\mathcal{S}$ and such that for every string $Y' \in \cup_i \mathcal{B}_i$ there is a string $Y \in \mathcal{A}$ and $P \in \mathsf{PAL}$ satisfying $Y' = Y\cdot P$. Each set is reported in representation of order at most t.

We first compute α, which takes $O(n)$ time by naive scanning. We then apply Theorem 10.5.2, which gives an affine set $\mathcal{B}$ in representation of order t in $O(t)$ time and space. Finally, we truncate $\mathcal{B}$ to prefixes of length $|XQ_1^\alpha|$ in $O(t^2)$ time and space with Algorithm 10.6.7, resulting in at most t representations of order at most t.

Algorithm 10.6.9 (Implementation of Corollary 10.5.5)**.** *Let $\langle X, (Q_i, \ell_i, u_i)_{i=1}^t\rangle$ be a canonical representation of an affine prefix set $\mathcal{A}$. There is a read-only algorithm that computes a decomposition of a superset of*

$$\mathcal{A}' = \{S\cdot P : S\cdot P \text{ is a prefix of } T, S \in \mathcal{A}, P \in \mathsf{PAL}, \mathrm{cen}(P) \geq |X| + (u_1+3)\cdot|Q_1|\}$$

(with the properties claimed in Corollary 10.5.5) into $O(t\log n)$ affine prefix sets of order at most $t+1$ in $O(n)$ time and $O(t^2\log n)$ space.

We start by searching the prefixes of $T[|XQ_1^{u_1+2}| + 1..]$ that belong to PAL. By Algorithm 10.6.2, we can find $O(\log n)$ affine sets of order at most 1 containing all such prefixes in $O(n)$ time and $O(\log n)$ space. The sets of order 1 are reported in strongly affine representation $\langle U(VU)^\ell, (VU, 1, u)\rangle$ for some $U \in \mathsf{PAL}\cup\{\varepsilon\}$, $V \in \mathsf{PAL}$ and integers $\ell \geq 1$ and $u > 1$.

As seen in Case 4 of the proof of Corollary 10.5.5, it cannot be that $|VU| < |Q_1|$. Also, as shown in Case 3, if $|VU| = |Q_1|$, then we can simply apply Algorithm 10.6.8, which takes $O(n)$ time and $O(t^2)$ space. Hence we only have to consider the cases where $|VU| > |Q_1|$ (Case 1), or where the affine prefix set is of order 0 (Case 2). For Case 1, let $\rho_1 = \langle XQ_1^{u_1+2}U(VU)^\ell, (VU, 1, u)\rangle$, and let $P' = U(VU)^{\ell+u}$. We obtain $\rho_2 =$

$\langle \mathsf{rev}(Q_1)[1..|Q_1|-s], (\mathsf{rev}(\hat{Q}_i), 1, u_i)_{i=1}^{t} \rangle$ in $O(t)$ time, where s and the $\hat{Q}_i$ are defined as in Theorem 10.5.4. Then, we compute the maximal $\alpha \in \mathbb{Q}$ such that $XQ_1^{u_1+2}P' \cdot \mathsf{rev}(Q_1)^\alpha$ is a prefix of T, which can be done naively in $O(n)$ time. Now we truncate ρ_2 such that the generated strings are of length at most $|Q_1^\alpha|$, which takes $O(t^2)$ time and space with Algorithm 10.6.7, and results in at most t representations of order at most t. Finally, we concatenate a copy of ρ_1 with each of these representations. In theory, when concatenating ρ_1 with some representation $\rho_2' = \langle X', (Q_i', \ell_i', u_i') \rangle$, we have to compute the primitive root Y of X'. This is because $(Y, \frac{|X'|}{|Y|}, \frac{|X'|}{|Y|})$ will become a component of the concatenation. However, we can omit this step by observing that the removal of fixed components with Algorithm 10.6.4 does not depend on the primitiveness of fixed components, and hence we can create and immediately remove a non-primitive component $(X', 1, 1)$ instead. Thus, we take $O(t^2)$ time per concatenation, or $O(t^3)$ time overall.We followed the computational steps in Case 1 of the proof of Corollary 10.5.5, and the total time and space complexity (for one affine set of core palindromes) are respectively $O(n + t^3)$ and $O(t^2)$.

As seen in the proof of Corollary 10.5.5, if an affine set of prefix-palindromes of $T[|XQ_1^{u_1+2}| + 1..]$ is given in representation $\langle P', \varepsilon \rangle$ (i.e., in Case 2), then we need a subset of the operations used for Case 1, leading to (at most) the same time and space complexity.

Summing over the $O(\log n)$ sets of core palindromes, the total time and space complexity are respectively $O(n \log n + t^3 \log n)$ and $O(t^2 \log n)$. This can be improved by observing that processing a single set of core palindromes takes $O(t^3)$ time if we ignore the $O(n)$ time needed to find the maximal $\alpha \in \mathbb{Q}$ such that $XQ_1^{u_1+2}P' \cdot \mathsf{rev}(Q_1)^\alpha$ is a prefix of T. We only use α to truncate ρ_2 to strings of length at most $|Q_1^\alpha|$, and every string generated by ρ_2 is of length over $|Q_1|$. Hence we only have to compute α if $\alpha > 1$. In Lemma 10.6.10 (below), we show how to perform this task in batch in overall $O(n)$ time, using $O(\log n)$ space. Hence the total time becomes $O(n + t^3 \cdot \log n)$, which is $O(n)$ due to Lemma 10.4.7.

Lemma 10.6.10. *Let $T[1..n]$ be a string and let Q be a fragment of T. Let $S_1, \ldots S_h$ be prefixes of T, in increasing order of length. There is a read-only algorithm that, in $O(n)$ time and $O(h)$ space, computes for each $i \in [1, h]$ either the maximal $\alpha \in \mathbb{Q}$ with $\alpha \geq 1$ such that $S_i \cdot \mathsf{rev}(Q)^\alpha$ is a prefix of T, or reports that such α does not exist.*

Proof. For $i \in [1, h]$, let $x_i = |S_i| + 1$. Using constant space and linear time pattern matching (see [78]), we enumerate all the occurrences of $\mathsf{rev}(Q)$ in T in increasing or. This makes it easy to filter out all the x_i for which $T[x_i..]$ does not have prefix $\mathsf{rev}(Q)$. Hence, from now on we can assume that all the $T[x_i..]$ have prefix $\mathsf{rev}(Q_1)$. Let $y_i \in [x_i + |Q|, n]$ be the maximal index such that $T[x_i..y_i]$ has period $|Q|$. We argue that, if for any $i \in [2, h]$ it holds $x_i \leq y_{i-1} - |Q| + 1$, then $y_{i-1} = y_i$. This is easy to see, because then $T[x_{i-1}..y_{i-1}]$ and $T[x_i..y_i]$ have overlap $T[x_i..y_{i-1}]$ of length at least $|Q|$, such that $T[x_{i-1}..y_{i-1}]$ and $T[x_i..y_i]$ must lie within a single $|Q|$-periodic fragment. For every $i \in [1, h]$ in increasing order, we compute y_i as follows. If $i = 1$ or $x_i > y_{i-1} - |Q| + 1$, then we naively scan $T[x_i + q..]$ until we reach the end position y_i of the $|Q|$-periodic fragment (recall that $T[x_i..]$ has prefix $\mathsf{rev}(Q)$ due to the earlier filtering). Otherwise, it holds $x_i \leq y_{i-1} - |Q| + 1$, and we can assign $y_i = y_{i-1}$ in constant time. We scan each position of T at most once (because a scan always starts at a position $x_i + q > y_{i-1}$), and thus the time is $O(n)$, while the space is $O(h)$. □

Recursively appending shorter palindromes

Algorithm 10.6.11 (Implementation of Lemma 10.5.7)**.** *Let $\mathcal{A}$ be an affine prefix set of $T[1..n]$, given in canonical representation ρ of order t. Define the set $\mathcal{S}$ as*

$$\mathcal{S} = \{S \cdot P \mid S \in \mathcal{A}, P \in \mathsf{PAL}, \text{ and } S \cdot P \text{ is a prefix of } T\}.$$

There is a read-only algorithm that, given ρ and T, computes $O((t+1)^2 \log n)$ affine prefix sets whose union is $\mathcal{S}$, using $O(n)$ time and $O((t+1)^3 \log n)$ space. Each affine prefix set is reported in representation of order at most $t+1$.

As explained in Lemma 10.5.7, we can build $O((t+1)^2 \log n)$ affine prefix sets such that their union equals $\mathcal{S}$ by a recursive application of Algorithm 10.6.8 and Algorithm 10.6.9. At every step of the recursion, the order of the affine set decreases by one, and hence the depth of the recursion is t. More precisely, the initial step at depth 0 takes $O(n)$ time and $O((t+1)^2 \log n)$ space by applying Algorithm 10.6.8 and Algorithm 10.6.9 to the original canonical representation. Then, at depth $i > 0$ of the recursion, we apply the algorithms to an affine prefix set of Q_i^3, and this set is in canonical representation of order $t-i$. Hence we use $O(|Q_i|)$ time and $O((t-i+1)^2 \cdot \log |Q_i|)$ space at depth i. This results in $O((t-i+1) \cdot \log |Q_i|)$ representations of order at most $(t-i+1)$. Then, each of these representations is appended to the same representation of order i, resulting in $O((t-i+1) \cdot \log |Q_i|)$ representations of order at most $t+1$, taking $O((t+1) \cdot (t-i+1) \cdot \log |Q_i|)$ time and space. Summing over all i, the space complexity is

$$O((t+1)^2 \cdot \log n + \sum_{i=1}^{t} (t+1) \cdot (t-i+1) \cdot \log |Q_i|) = O((t+1)^3 \log n).$$

The time complexity is the same, with an additional $O(n + \sum_{i=1}^{t} |Q_i|)$ needed. Since the representation is canonical, Lemma 10.4.6 implies $|Q_1| > \sum_{i=2}^{t} |Q_i|$ and thus $O(n + \sum_{i=1}^{t} |Q_i|) = O(n)$. Note that $O(n)$ also dominates $O((t+1)^3 \log n)$ due to Lemma 10.4.7.

Implementation of Theorem 10.1.1

The final component of the algorithm is a for-loop that goes over all $1 \leq i \leq k$ to construct a collection $\mathcal{C}_i$ of affine sets containing all prefixes of T that belong to PAL^i, where each affine set has canonical representation of order at most i. To construct $\mathcal{C}_1$ with $|\mathcal{C}_1| = O(\log n)$, we use Algorithm 10.6.2 in $O(n)$ time and $O(\log n)$ space.

Now we inductively show how to compute $\mathcal{C}_i$ with $i > 1$ from $\mathcal{C}_{i-1}$. For each affine prefix set in $\mathcal{C}_{i-1}$, or more precisely for its canonical representation (necessarily of order at most $i-1$), we proceed as follows. We apply Algorithm 10.6.11 to append a palindrome, which takes $O(|\mathcal{C}_{i-1}| \cdot n)$ time and $O(|\mathcal{C}_{i-1}| \cdot i^3 \log n)$ space in total, and results in $O(|\mathcal{C}_{i-1}| \cdot i^2 \cdot \log n)$ representations of order at most i. Next, we have to make the new representations canonical. We apply Algorithm 10.6.6 to each of them, which takes $O(6^i \cdot i^2)$ time and $O(6^i \cdot i)$ space per representation. The total time is $O(|\mathcal{C}_{i-1}| \cdot i^4 \cdot \log n \cdot 6^i)$, while the total space is $O(|\mathcal{C}_{i-1}| \cdot i^3 \cdot \log n \cdot 6^i)$. The result are $O(|\mathcal{C}_{i-1}| \cdot i^2 \cdot \log n \cdot 6^i)$ canonical representations, each of which is of order at most i. Finally, $\mathcal{C}_i$ consists exactly of the sets represented by these representations.

Now we analyze the total time and space complexity. The algorithm terminates after computing $\mathcal{C}_k$, where k is the palindromic length of T. We focus on a fixed $i \in [2, t]$ and analyze the complexity of computing $\mathcal{C}_i$. If we ignore the $O(|\mathcal{C}_{i-1}| \cdot n)$ time needed to append a palindrome, then the time and space for computing $\mathcal{C}_i$, and also its cardinality $|\mathcal{C}_i|$ are bounded from above by some value $U_i = O(|\mathcal{C}_{i-1}| \cdot i^4 \cdot \log n \cdot 6^i)$. Hence we can

find a constant $c \geq 2$ independent of i and n such that $U_i = c \cdot |\mathcal{C}_{i-1}| \cdot i^4 \cdot \log n \cdot 6^i$. This also holds for $\mathcal{C}_1$ if we define $U_0 = |\mathcal{C}_0| = 1$, and hence we consider $i \in [1, k]$ from now on. We resolve the recurrence and obtain the bound

$$\begin{aligned} U_i &= c \cdot |\mathcal{C}_{i-1}| \cdot i^4 \cdot \log n \cdot 6^i \\ &\leq c \cdot U_{i-1} \cdot i^4 \cdot \log n \cdot 6^i \\ &\leq \prod_{j=1}^{i} c \cdot j^4 \cdot 6^j \cdot \log n \\ &< c^i \cdot (i!)^4 \cdot 6^{i(i+1)/2} \cdot \log^i n. \end{aligned}$$

Let $\epsilon \in \mathbb{R}^+$ with $\epsilon < 2$ be an arbitrary constant, and let $\epsilon' = \frac{1}{2-\epsilon} - \frac{1}{2} - \frac{1}{2c} > 0$ for c large enough. If $i \leq c$ or $(i^i)^6 \geq 6^{\epsilon' \cdot i^2}$, then i is constant and it holds $U_i = O(\log^i n) = O(6^{i^2/(2-\epsilon)}/i \cdot \log^i n)$. Otherwise, $i > c \geq 2$ implies $i < c^i < i^i$ and thus $c^i \cdot (i!)^4 < (i^i)^6/i < 6^{\epsilon' \cdot i^2}/i$. We continue with

$$\begin{aligned} U_i &< c^i \cdot (i!)^4 \cdot 6^{i(i+1)/2} \cdot \log^i n \\ &< (6^{\epsilon' \cdot i^2}/i) \cdot 6^{i(i+1)/2} \cdot \log^i n \\ &= (6^{i^2/(2-\epsilon)}/i) \cdot \log^i n \end{aligned}$$

We have shown $U_i = O((6^{i^2/(2-\epsilon)}/i) \cdot \log^i n)$. By summing over all $i \in [1, t]$, we obtain

$$\begin{aligned} \sum_{i=1}^{k} U_i &= O(\sum_{i=1}^{k} (6^{i^2/(2-\epsilon)}/i) \cdot \log^i n) \\ &\leq O(\sum_{i=1}^{k} (6^{k^2/(2-\epsilon)}/k) \cdot \log^k n) \\ &= O(6^{k^2/(2-\epsilon)} \cdot \log^k n), \end{aligned}$$

where the middle step follows from the fact that $(6^{i^2/(2-\epsilon)}/i) \cdot \log^i n \leq (6^{k^2/(2-\epsilon)}/k) \cdot \log^k n$ for arbitrary $n \in \mathbb{N}^+$ and $i \in [1, k]$. We have shown that the total space for computing all $\mathcal{C}_i$ with $i \in [1, k]$ is $O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$. For the time bound, we ignored the $O(|\mathcal{C}_{i-1}| \cdot n)$ time needed to compute $\mathcal{C}_i$. However, we already know that $\sum_{i=1}^{k} |\mathcal{C}_i| < \sum_{i=1}^{k} U_i = O(6^{k^2/(2-\epsilon)} \cdot \log^k n)$, and thus the time is bounded by $O(n \cdot 6^{k^2/(2-\epsilon)} \cdot \log^k n)$. □

10.6.1 Computing the palindromic length

The algorithm of Theorem 10.1.3 can be used to test if the palindromic length of T is at most k by checking whether T is a k-palindromic prefix. We now show how to improve the complexity by using two copies of the data structure of Theorem 10.1.3.

Theorem 10.1.4. *Given a string T of length n, there is a read-only algorithm that computes the palindromic length k of T in $O(n \cdot 6^{k^2} \cdot \log^{\lceil k/2 \rceil} n)$ time and $O(6^{k^2} \cdot \log^{\lceil k/2 \rceil} n)$ space.*

Proof. Let $0 < \epsilon < 1$ be a constant. We will start with $k = 1$, and increment k until we can verify that the palindromic length is indeed k. We will consider strings in $\mathsf{PAL}^{\lfloor k/2 \rfloor}$ and $\mathsf{PAL}^{\lceil k/2 \rceil}$, and during the complexity analysis we encounter terms of the form $6^{\lceil k/2 \rceil^2/(2-\epsilon)}$.

Before we describe the algorithm, we point out that these terms can be bounded from above using

$$6^{\lceil k/2\rceil^2/(2-\epsilon)} \leq 6^{((k+1)/2)^2/(2-\epsilon)} = 6^{(k^2+2k+1)/(8-4\epsilon)} < 6^{(k^2)/(8-4\epsilon)} \cdot 6^{3k} < 6^{k^2/(8-6\epsilon)},$$

where the last step holds if k exceeds a constant that depends solely on ϵ.

The algorithm calls Theorem 10.1.3 to build a set $\mathcal{P}$ that contains $O(6^{k^2/(8-6\epsilon)} \cdot \log^{\lceil k/2\rceil} n)$ affine prefix sets that describe the $\lceil k/2\rceil$-palindromic prefixes of T. It then calls Theorem 10.1.3 again to compute the set $\mathcal{S}$ that contains $O(6^{k^2/(8-6\epsilon)} \cdot \log^{\lfloor k/2\rfloor} n)$ affine prefix sets of $\mathsf{rev}(T)$ that describe the $\lfloor k/2\rfloor$-palindromic prefixes of $\mathsf{rev}(T)$. Each set is given in canonical representation of order at most $\lceil k/2\rceil$ for $\mathcal{P}$, and at most $\lfloor k/2\rfloor$ for $\mathcal{S}$. The required time is $O(n \cdot 6^{k^2/(8-6\epsilon)} \cdot \log^{\lceil k/2\rceil} n)$, and the overall space is $O(6^{k^2/(8-6\epsilon)} \cdot \log^{\lceil k/2\rceil} n)$.

Then, we iterate over each possible combination of a set $\mathcal{A} \in \mathcal{P}$ and a set $\mathcal{B} \in \mathcal{S}$. We use Lemma 10.6.12 (below) to check whether there exists $A \in \mathcal{A}$ and $B \in \mathcal{B}$ such that $T = A \cdot \mathsf{rev}(B)$. Then, T is in PAL^k if and only if at least one of these checks is succesful. There are $O(6^{k^2/(4-3\epsilon)} \cdot \log^k n)$ possible combinations, and each combination can be verified in $O(10^k)$ time. Hence, the total time needed for verification is $O(10^k \cdot 6^{k^2/(4-3\epsilon)} \cdot \log^k n)$.

For the final part of the analysis, recall that we have to increment k and rerun the algorithm until the verification is successful. From now on, let k be the actual palindromic length of T. The space is dominated by the final run, which requires $O(6^{k^2/(8-6\epsilon)} \cdot \log^{\lceil k/2\rceil} n)$ space. The total time is bounded by

$$\begin{aligned} & O(k \cdot n \cdot 6^{k^2/(8-6\epsilon)} \cdot \log^{\lceil k/2\rceil} n + k \cdot 10^k \cdot 6^{k^2/(4-3\epsilon)} \cdot \log^k n) \\ \leq\ & O(\quad n \cdot 6^{k^2/(8-7\epsilon)} \cdot \log^{\lceil k/2\rceil} n \quad + \quad 6^{k^2/(4-4\epsilon)} \cdot \log^k n) \\ \leq\ & O(\quad n \cdot 6^{k^2} \cdot \log^{\lceil k/2\rceil} n \quad + \quad 6^{k^2} \cdot \log^k n), \end{aligned}$$

where we used the same trick as before to hide the factors k and 10^k by increasing the coefficient of ϵ. If $k \leq \sqrt{\log_6 n}$, then $\log^k n = o(n)$ and the time complexity is clearly dominated by the term $n \cdot 6^{k^2} \cdot \log^{\lceil k/2\rceil} n$. If $k > \sqrt{\log_6 n}$, then $6^{k^2} > n$ and both time and space are superlinear. Hence, when running the algorithm, we stop increasing k as soon as it exceeds $\sqrt{\log_6 n}$. If we terminate before, i.e., if the palindromic length is less than $\sqrt{\log_6 n}$, then we achieve the claimed time complexity. Otherwise, i.e., if we terminate because k exceeds $\sqrt{\log_6 n}$, we finish the computation using the algorithm by Borozdin et al. [76], which takes $O(n) \leq O(6^{k^2})$ time and space. □

Lemma 10.6.12. *Let $\mathcal{A}$ be an affine prefix set of T, and let $\mathcal{B}$ be affine prefix set $\mathsf{rev}(T)$. If $\mathcal{A}$ and $\mathcal{B}$ are given in canonical representation of orders respectively at most t and t', then we can decide if there are $A \in \mathcal{A}$ and $B \in \mathcal{B}$ with $T = A \cdot \mathsf{rev}(B)$ in $O(10^{t+t'})$ time and space.*

Proof. Let $\rho = \langle X, (Q_i, 1, u_i)_{i=1}^{t}\rangle$ and $\rho' = \langle X', (Q'_i, 1, u'_i)_{i=1}^{t'}\rangle$ be the respective canonical representations of $\mathcal{A}$ and $\mathcal{B}$. We implement the procedure recursively. If $t = t' = 0$, then we can check in constant time if the lengths of the two generated strings sum to n. If $t > 0$ and $t' = 0$ (the case $t = 0$ and $t' > 0$ is symmetric), then $\mathcal{B}$ contains a single string of some length m, and we only have to check if $\mathcal{A}$ contains a string of length $n' = n - m$. If $|XQ_1Q_2Q_3 \dots Q_t| > n'$, then every string generated by $\mathcal{A}$ is of length over n', and we can terminate with a negative answer. Otherwise, let

$$a_{\min} = (n' - |X| - |Q_2^{u_2}Q_3^{u_3} \dots Q_t^{u_t}|)/|Q_1| \quad \text{and} \quad a_{\max} = (n' - |X| - |Q_2Q_3 \dots Q_t|)/|Q_1|$$

(both in $\mathbb{Q}$), and note that a string $XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t}$ with $\forall i \in [1,t] : a_i \in [1,u_i]$ can only be of length n' if $a_{\min} \le a_1 \le a_{\max}$. By Corollary 10.4.16, it holds $a_{\max} - a_{\min} < 1$, hence there is at most one $a \in \mathbb{N}$ such that $a_{\min} \le a \le a_{\max}$. If $a \in [1,u_1]$, then we replace ρ with $\langle XQ_1^a, (Q_i,1,u_i)_{i=2}^t\rangle$ and recurse. Otherwise, we terminate with a negative answer.

It remains the most general case $t > 0$ and $t' > 0$. If there are $A \in \mathcal{A}$ and $B \in \mathcal{B}$ such that $T = A \cdot \mathsf{rev}(B)$, then it holds $A = XQ_1^{a_1}Q_2^{a_2}\dots Q_t^{a_t}$ and $B = X'Q_1'^{a_1'}Q_2'^{a_2'}\dots Q_{t'}'^{a_{t'}}$ for some exponents satisfying $\forall i \in [1,t] : a_i \in [1,u_i]$ and $\forall i \in [1,t'] : a_i' \in [1,u_i']$. For now, assume $|Q_1| = |Q_1'|$. Let $a = \min(u_1 - a_1, a_1' - 1)$, then the representations generate strings such that

$$T = XQ_1^{a_1+a}Q_2^{a_2}\dots Q_t^{a_t} \;\cdot\; \mathsf{rev}(X'Q_1'^{a_1'-a}Q_2'^{a_2'}\dots Q_{t'}'^{a_{t'}}).$$

Note that either $a_1 + a = u_1$ or $a_1' - a = 1$ (or both). We proceed with two recursive calls. In the first one, we replace ρ with $\langle XQ_1^{u_1}, (Q_i,1,u_i)_{i=2}^t\rangle$. In the second one, we replace ρ' with $\langle X'Q_1', (Q_i',1,u_i')_{i=1}^{t'}\rangle$. If both recursive calls have negative answer, then we terminate with negative answer. Otherwise, we terminate with a positive answer.

Now we can assume $t > 0$, $t' > 0$, and $|Q_1| \ne |Q_1'|$. We only consider $|Q_1| > |Q_1'|$, as the other case is symmetric. We again assume that there are $A \in \mathcal{A}$ and $B \in \mathcal{B}$ such that $T = A \cdot \mathsf{rev}(B)$, where A and B are defined as before. Since a canonical representation is strongly affine, $A' = XQ_1^{a_1+2}Q_2^{a_2}\dots Q_t^{a_t}$ is a prefix of T. By Corollary 10.4.16 and Lemma 10.4.5, it is clear that $A = XQ_1^{a_1}Q_1[1..s]$ and $A' = XQ_1^{a_1+2}Q_1[1..s] = A \cdot (\mathsf{rot}^s(Q_1))^2$ for some $s \in [1,|Q_1|)$. If $|A'| < n - |X'|$, then the primitive square $(\mathsf{rot}^s(Q_1))^2$ is not only a prefix of $\mathsf{rev}(B) = \mathsf{rev}(X'(Q_1')^{a_1'}(Q_2')^{a_2'}\dots(Q_{t'}')^{a_{t'}})$, but also a prefix of $B' = \mathsf{rev}((Q_1')^{a_1'}(Q_2')^{a_2'}\dots(Q_{t'}')^{a_{t'}})$. However, by Lemma 10.4.5, we know that B' and thus also $(\mathsf{rot}^s(Q_1))^2$ has period $|Q_1'| < |Q_1|$, which contradicts Lemma 10.2.1. Hence we have shown that $|A'| \ge n - |X'|$, which implies

$$a_1 \ge (n - |X'| - |X| - |Q_2^{a_2}Q_3^{a_3}\dots Q_t^{a_t}|)/|Q_1| - 2 > (n - |X'| - |X|)/|Q_1| - 3,$$

where the second inequality is due to Corollary 10.4.16. We define

$$a_{\min} = (n - |X| - |X'|)/|Q_1| - 3 \qquad \text{and} \qquad a_{\max} = (n - |X| - |X'|)/|Q_1|$$

(both in $\mathbb{Q}$), and observe that $a_1 < a_{\max}$ because otherwise $|A| \ge n - |X'|$ and thus $|A \cdot \mathsf{rev}(B)| > n$. We have established $a_{\min} < a_1 < a_{\max}$. It holds $a_{\max} - a_{\min} = 3$, which means that there are at most three possible $a \in \mathbb{N}$ such that $a_{\min} < a < a_{\max}$. For each $a \in [1,u_1]$ with $a_{\min} < a < a_{\max}$, we recurse by replacing ρ with $\langle XQ_1^a, (Q_i,1,u_i)_{i=2}^t\rangle$. If all of the at most three recursive calls have a negative answer, we terminate with a negative answer. Otherwise, we terminate with a positive answer.

Regardless of the case, we perform at most three recursive calls, and with each call the combined order of the representations decreases by one. Thus, the total number of calls is at most $3^{t+t'}$. In each call, computing the new representations takes $O(t+t')$ time with a naive implementation (only simple arithmetic operations are needed). At all times, the required space is linear in the time spent. Hence the total time and space are $O(3^{t+t'} \cdot (t+t'))$, which is less than $O(10^{(t+t')/2})$ if $t + t'$ exceeds a sufficiently large constant. □

Conclusion and Perspectives

In this thesis, we study problems related to approximate pattern matching and approximate language membership, for different notions of approximation: string distances, property testing, and others. Chapter 2 introduces the concepts used in this thesis.

In the first part of this thesis, we present algorithms and data structures for approximate pattern matching tasks. In Chapter 4 and Chapter 6, we give data structures for operations that are crucial for approximate pattern matching (as observed by Charalampopoulos et al. [89]), with a focus on time-space trade-offs. Chapter 4 gives a small-space data structure for internal pattern matching, which we then use to give algorithms for circular pattern matching and longest common substring in low-memory settings. In Chapter 6, we give a data structure for longest common extension in string with wildcards that offers a time-space trade-off that interpolates smoothly between the solution of Crochemore et al. [117] and the kangaroo jumping method of Landau and Vishkin [224]. We show that this data structure allows faster algorithms for approximate pattern matching and analysis of strings with wildcards. In Chapter 5, we study approximate pattern matching under the Hamming distance in strings with wildcards. We give an efficient algorithm for the low-distance regime, in the case when there are few contiguous groups of wildcards in the input strings. We further use our algorithm to give insight on the structure of the occurrences of a pattern in a text as a small number of arithmetic progressions in this setting, and give an almost-tight lower bound for this characterization.

In Part II, we give algorithms for deciding approximate membership in formal languages. In Chapter 8, we give a complete characterization of the complexity of property testing of regular languages under the Hamming distance. Our results close a line of work opened by Alon et al. [24]: we not only find the exact (asymptotic) complexity of the problem, but also outline three complexity classes, show that there are no other classes, and give a combinatorial characterization of each class. Furthermore, we show that our characterization is decidable: it is complete for PSPACE. In Chapter 9, we give streaming and read-only algorithms for computing the distance to the languages of palindromes or squares under the Hamming and edit distances, in the low-distance regime. Our algorithms rely on approximate pattern matching algorithms similar to those discussed in Part I and on distance sketches [62, 107]. Finally, in Chapter 10, we consider the problem of computing the palindromic length of a string. We first establish results on the structure of prefixes of palindromic length k of a string. We then show that this structure can be computed efficiently in small space, leading to a space-efficient algorithm for computing the palindromic length in the low-distance regime.

Next, we discuss some open questions and research directions related to the topics covered in this thesis.

Longest common extension with mismatches

Our study of longest common extension with wildcards, presented in Chapter 6, originally started as a study of longest common extension with *mismatches* in the low-distance regime, with the goal of giving faster algorithms for pattern matching with mismatches. The kangaroo jumping method of Landau and Vishkin [224] gives a data structure that solves this last problem using $O(n)$ preprocessing time and $O(k)$ query time. In the case of wildcards, we can use additional preprocessing to make queries faster, as shown by Crochemore et al. [117] and trade-off given in Chapter 6. A natural question is whether the same is true for LCE with mismatches.

Open Question 1. Can longest common extension with mismatches be solved in $O(n \cdot t)$ preprocessing time and $O(k/t)$ query time for $t = \Omega(k^\epsilon)$ with $\epsilon > 0$?

Note that for the edit distance, which can be computed in time $O(k^2)$ after $O(n)$ time preprocessing, such an improvement is ruled out by conditional lower bounds.

Property testing and variants for context-free languages

In Chapter 8, we give a complete characterization of the complexity of property testing of regular languages, which includes an upper bound of $O(\log(\epsilon^{-1})/\epsilon)$ queries. A natural question is: what is the complexity of property testing of other classes of languages, such as context-free languages? Alon et al. [24, Theorem 2] showed that context-free languages cannot be tested with a constant number of queries: they give an explicit context-free language that requires $\Omega(\sqrt{n})$ queries.

Two main research directions arise from this observation. The first focuses on understanding the complexity of property testing of context-free languages of interest. An important family of context-free languages are the DYCK languages, the sets of well-parenthesized expressions with one or more kinds of parentheses, as they capture the ability of context-free languages to express hierarchical information (this idea is formalized by the Chomsky-Schützenberger Representation Theorem [99]). Fischer et al. [136] gave an algorithm that uses $O(n^{2/5})$ queries for testing DYCK languages, and show a lower bound of $\Omega(n^{1/5})$ queries for this problem.

Open Question 2. What is the complexity of property testing of DYCK languages?

The second line of research observes that context-free languages are also hard for the streaming model, and proposes to study membership in the hybrid *streaming property testing* model, a framework that combines the sequential access model of streaming with the promise of property testing that the input is either in the language or far from it. The goal is the same as in the streaming model: use as little space as possible. François et al. [142] studied streaming property testing of visibly pushdown languages (VPLs), a strict subclass of context-free languages that contains the DYCK languages, is hard for streaming and has strong closure properties. The algorithm of François et al. [142] reduces to property testing of the regular language of "folded" peak-shaped words. Their reduction maps to weighted edit distance case, for which they design a property tester that uses $O(1/\epsilon^2)$ queries, and their algorithm for VPLs uses $O(\log^6 n/\epsilon^4)$ bits of space. In [48], we show that the property testing algorithm presented in Chapter 8 also works in the case of the weighted edit distance. Using our improved tester, we give an improved algorithm for streaming property testing of VPLs that uses $O(\log^5 n \log\log n/\epsilon^3)$ bits of space. We also give a lower bound of $\Omega(\max(\log n, 1/\epsilon))$ bits of space. There is still a large gap between these upper and lower bounds.

Open Question 3. What is the complexity of streaming property testing of Visibly Pushdown Languages?

Furthermore, the context-free language used by Alon et al. [24] for the property testing lower bound is *not* a VPL. This leads to the question of the complexity of property testing of VPLs, and whether the Dyck languages are the hardest VPLs.

Open Question 4. What is the complexity of property testing of VPLs? Are there VPLs that are harder to test than Dyck languages?

In Chapter 8, property testing of regular languages is reduced to finding occurrences of blocking sequences, which are obstructions to membership in the language. Consequently, I believe that an answer to all three of the above open questions requires insight on the nature of obstructions for VPLs and Dyck languages.

Open Question 5. What is the nature of obstructions for VPLs and Dyck languages? How can we find them efficiently with (streaming) sampling?

Practical aspects of approximate pattern matching

In general, this thesis studies algorithms and data structures from a rather *theoretical* point of view, with an emphasis on minimizing asymptotic complexity and understanding the underlying mathematical objects. However, our approach does not take into consideration ease of implementation or practical performance. On the other hand, modern search engines that allow approximate search, such as Meilisearch [239], are based on heuristic approaches that are not efficient in theory but work well in practice. I believe that the theoretical and practical approaches are not mutually exclusive, and that aspects of one can be used to improve the other.

Open Question 6. How can we bridge the gap between theory and practice of approximate pattern matching?

Part III

Appendix

Appendix A

Unrelated Work

This section reviews some additional research work that I conducted during the three years of Ph.D., but which is not directly relevant to the topic of my thesis.

Towards stronger depth lower bounds

This thesis puts a particular emphasis on giving lower bounds that are as close as possible to matching the complexity of algorithms. However, it often seems much to prove a lower bound than it is to give an algorithm. The field of lower bounds in complexity is still vastly unexplored: for example, we know that there are functions of complexity $\Omega(n^c)$ for any $c > 0$ in P, however these results are obtained through diagonalization and are not constructive. On the other hand, the best lower bound that we know against an *explicit* function in P is $\Omega(n^{3-\epsilon})$ for any $\epsilon > 0$, due to Håstad [173].

With Ryan Williams, we explored two ways of giving better lower bounds against explicit functions. One approach builds on the hypothesis that there exists algorithms for SAT that are marginally faster than the current state of the hard, and explicitly build a hard function from this assumption. The second approach give depth lower bounds of $3.603 \log n$ for uniform circuits computing SAT. This model is slightly weaker than polynomial time algorithms, but improves what was known for depth lower bounds, as an $\Omega(n^c)$ time lower bound implies a depth lower bound of $c \log n + \Omega(1)$. The resulting paper [3] was published in ITCS'24.

An Approximation scheme for Ultrametric Embedding

Ultrametrics are a mathematical concept that provide a rigorous foundation for the study of *hierarchical clustering*: the problem of computing the best hierarchical clustering of a metric space $(\mathcal{S}, d)$ of n points is equivalent to that of computing the ultrametric Δ that minimizes the distortion w.r.t. d. A case of particular interest is the Euclidean case, i.e. when $\mathcal{S}$ is a subset of $\mathbb{R}^t$ and the distance d is the Euclidean metric ℓ_2. In this setting, Farach et al. [131] gave a quadratic-time algorithm that computes an ultrametric minimizing the worst-case distortion, and proved that the problem cannot be solved in sub-quadratic time. This quadratic runtime is prohibitive for modern applications that need to handle very large datasets. To circumvent this lower bound, Cohen-Addad et al. [109] and later Cohen-Addad et al. [110] gave approximation algorithms for the problem that produce an ultrametric approximating the minimum distortion within a factor of $5c$ and $\sqrt{2}c$ for any $c > 1$, respectively, while running in time $\tilde{O}(n^{1+O(1/c^2)})$.

However, the constant hidden in the $O(\cdot)$ in the exponent of the runtime is at least 12, which makes the algorithm impractical.

Together with Guillaume Lagarde, we gave the first approximation algorithm with an approximation ratio c arbitrarily close to 1, while running in sub-quadratic time $\tilde{O}(n^{1+1/c^2})$. In our contribution, we give an improved algorithm for computing γ-Kruskal trees in Euclidean graphs, a combinatorial object of independent interest, thereby removing the large constant in the exponent of the running time and making the algorithm truly sub-quadratic for all values of $c > 1$. We also give an efficient dynamic data structure for computing approximate farthest neighbors in Euclidean metric spaces. Finally, we provide a Rust implementation that shows that when $c \simeq \sqrt{2}$, the performance of our algorithm is comparable to that of previous work, and that it scales to large datasets. This work [1] was published at the AAAI 2025 conference.

Synthesis of LTL_f formulas

Program synthesis is the task of finding a program $\mathcal{P}$ that maps a given set of inputs $\{X_i\}$ to their corresponding output $\{Y_i\}$ and minimizes a given cost function.

With Nathanaël Fijalkow, Théo Matricon, Baptiste Mouillon and Pierre Vandenhove, we studied the case of synthesizing LTL formulas over finite words (in short, LTL_f formulas) when given a set of positive and a set of negative inputs, aiming for the smallest possible formula. We use an enumerative approach, that lists all formulas in order of increasing size, and stops when it finds one that satisfies the input. We filter out formulas equivalent to previously seen formulas to speed up the search. When the number of enumerated formulas becomes too large, our algorithm switches to enumerating boolean combinations of LTL_f formulas. This allows use to filter formulas that are *dominated* by others, and greatly reduces the size of the search space. We provided a Rust implementation that demonstrates the practical performance of our algorithm.

Constant-delay enumeration of regular languages under the Hamming distance.

Amarilli and Monet [28] studied the task of enumerating a regular language L, that is, producing the (usually infinite) sequence of its words, while bounding the *delay* between two consecutive words, i.e. allowing only a constant number of operations when producing one word from the previous. They give an effective characterization of languages that can be enumerated with constant delay, and give an algorithm using *edit* operations, which induces a total order on the words such that the *edit distance* between a word and the next is bounded.

With Antoine Amarilli and Mikaël Monet, we studied the restriction of the problem to the more constraining Hamming distance, where only substitutions are allowed. We extend their results: we give an effective characterization of languages that can be enumerated with constant delay, and give a construction for a total order on the words where the Hamming distance between a word and the next is bounded by a constant.

Parameterized Algorithms and Computational Experiments (PACE) Challenges

The PACE Challenge[1] is an annual programming contest that "aims to bridge the gap between the theory and practice of algorithm design and engineering". Each year, the target is a different problem from the theory of algorithms, and teams of contestants must implement the fastest possible solution for the problem. During my Ph.D., I participated in two editions of the challenge.

- In 2022, the problem to solve was Directed Feedback Vertex Set. With Gaétan Berthe, Yoann Coudert-Osmont, David Desobry, Amadeus Reinald and Mathis Rocton, we wrote a solver that reached the 2nd position on the Heuristic track. We were subsequently invited to publish a description of our solver at IPEC 2022 [4].
- In 2023, the goal was computing the twin-width of graphs. With Jérôme Boillot, Nicolas Bousquet and Théo Pierron, we wrote a solver that reached the 6th position on the Exact track.

[1] https://pacechallenge.org/

List of Figures

List of Tables

List of Algorithms

List of Publications

[1] Gabriel Bathie and Guillaume Lagarde. A $(1+\epsilon)$-approximation for ultrametric embedding in subquadratic time. *Proc. of AAAI'25*, 39(15):15516–15523, apr 2025. ISSN 2159-5399. doi: 10.1609/aaai.v39i15.33703. URL http://dx.doi.org/10.1609/aaai.v39i15.33703.

[2] Gabriel Bathie and Tatiana Starikovskaya. Property testing of regular languages with applications to streaming property testing of visibly pushdown languages. In *Proc. of ICALP*, volume 198 of *LIPIcs*, pages 119:1–119:17, 2021. doi: 10.4230/LIPICS.ICALP.2021.119. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.119.

[3] Gabriel Bathie and R. Ryan Williams. Towards Stronger Depth Lower Bounds. In *Proc. of ITCS*, volume 287 of *LIPIcs*, pages 10:1–10:24, 2024. ISBN 978-3-95977-309-6. doi: 10.4230/LIPIcs.ITCS.2024.10. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITCS.2024.10.

[4] Gabriel Bathie, Gaétan Berthe, Yoann Coudert-Osmont, David Desobry, Amadeus Reinald, and Mathis Rocton. PACE Solver Description: DreyFVS. In *Proc. of IPEC*, volume 249 of *LIPIcs*, pages 31:1–31:4, 2022. ISBN 978-3-95977-260-0. doi: 10.4230/LIPIcs.IPEC.2022.31. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.IPEC.2022.31.

[5] Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya. Small-Space Algorithms for the Online Language Distance Problem for Palindromes and Squares. In *Proc. of ISAAC*, pages 10:1–10:17, 2023. doi: 10.4230/LIPICS.ISAAC.2023.10.

[6] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Longest Common Extensions with Wildcards: Trade-off and Applications. In *Proc. of ESA*, 2024.

[7] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Internal Pattern Matching in Small Space and Applications. In *Proc. of CPM*, volume 296, pages 4:1–4:20, 2024. ISBN 978-3-95977-326-3. doi: 10.4230/LIPIcs.CPM.2024.4. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2024.4.

[8] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern Matching with Mismatches and Wildcards. In *Proc. of ESA*, volume 308, pages 20:1–20:15, 2024. ISBN 978-3-95977-338-6. doi: 10.4230/LIPIcs.ESA.2024.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2024.20.

[9] Gabriel Bathie, Jonas Ellert, and Tatiana Starikovskaya. Small space encoding and recognition of k-palindromic prefixes, 2025. URL https://arxiv.org/abs/2410.03309.

References

[10] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Proc. of ICALP*, pages 39–51, 2014. doi: 10.1007/978-3-662-43948-7_4. (cit. on p. 64).

[11] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the Current Clique Algorithms Are Optimal, so Is Valiant's Parser. *SIAM Journal on Computing*, 47(6):2527–2555, 2018. doi: 10.1137/16M1061771. (cit. on pp. xv, 4, 84, and 127).

[12] Amir Abboud, Karl Bringmann, Nick Fischer, and Marvin Künnemann. The time complexity of fully sparse matrix multiplication. In *Proc. of SODA*, pages 4670–4703, 2024. doi: 10.1137/1.9781611977912.167. (cit. on p. 66).

[13] Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13 (11), 2020. ISSN 1999-4893. doi: 10.3390/a13110276. (cit. on p. 23).

[14] Karl R. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987. (cit. on pp. 16, 39, and 41).

[15] Peyman Afshani and Jesper Sindahl Nielsen. Data structure lower bounds for document indexing problems. In *Proc. of ICALP*, pages 93:1–93:15, 2016. doi: 10.4230/LIPICS.ICALP.2016.93. (cit. on p. 64).

[16] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. ISSN 0001-0782. doi: 10.1145/360825.360855. (cit. on pp. xii and 2).

[17] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1974. ISBN 0201000296. (cit. on p. 125).

[18] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972. doi: 10.1137/0201022. (cit. on pp. 84 and 127).

[19] Alfred V. Aho, Monica S. Lam, and Jeffrey D. Ullman. *Compilers principles, techniques & tools*. Pearson Education, 2007. (cit. on pp. xiv and 4).

[20] Tatsuya Akutsu. Approximate string matching with don't care characters. *Inf. Process. Lett.*, 55(5):235–239, 1995. doi: 10.1016/0020-0190(95)00111-O. URL https://doi.org/10.1016/0020-0190(95)00111-O. (cit. on pp. 17, 22, 43, 63, 64, 66, and 76).

[21] Noga Alon and Michael Krivelevich. Testing k-colorability. *SIAM Journal on Discrete Mathematics*, 15(2):211–227, 2002. (cit. on pp. 81 and 82).

[22] Noga Alon and Asaf Shapira. Testing subgraphs in directed graphs. In *Proc. of STOC*, pages 700–709, 2003. (cit. on p. 81).

[23] Noga Alon and Asaf Shapira. Every monotone graph property is testable. In *Proc. of STOC*, page 128–137, 2005. ISBN 1581139608. doi: 10.1145/1060590.1060611. URL https://doi.org/10.1145/1060590.1060611. (cit. on p. 82).

[24] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. Regular languages are testable with a constant number of queries. *SIAM Journal on Computing*, 30(6):1842–1862, 2001. doi: 10.1109/SFFCS.1999.814641. (cit. on pp. xv, 4, 81, 82, 83, 89, 90, 91, 92, 94, 100, 109, 119, 120, 121, 193, 194, and 195).

[25] Noga Alon, Eldar Fischer, Ilan Newman, and Asaf Shapira. A combinatorial characterization of the testable graph properties: it's all about regularity. In *Proc. of STOC*, pages 251–260, 2006. (cit. on p. 82).

[26] Noga Alon, Tali Kaufman, Michael Krivelevich, and Dana Ron. Testing triangle-freeness in general graphs. *SIAM Journal on Discrete Mathematics*, 22(2):786–819, 2008. (cit. on pp. 81 and 82).

[27] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. of STOC*, STOC '04, page 202–211, 2004. ISBN 1581138520. doi: 10.1145/1007352.1007390. URL https://doi.org/10.1145/1007352.1007390. (cit. on p. 163).

[28] Antoine Amarilli and Mikaël Monet. Enumerating regular languages with bounded delay. *arXiv preprint arXiv:2209.14878*, 2022. (cit. on p. 200).

[29] Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. In *Proc. of ICALP*, volume 198 of *LIPIcs*, pages 116:1–116:17, 2021. doi: 10.4230/LIPICS.ICALP.2021.116. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.116. (cit. on p. 90).

[30] Andris Ambainis. Quantum query algorithms and lower bounds. In *Classical and New Paradigms of Computation and their Complexity Hierarchies*, pages 15–32, 2004. doi: 10.1007/978-1-4020-2776-5_2. (cit. on p. 59).

[31] Amihood Amir and Benny Porat. Approximate on-line palindrome recognition, and applications. In *Proc. of CPM*, volume 8486 of *LNCS*, pages 21–29, 2014. doi: 10.1007/978-3-319-07566-2_3. (cit. on pp. 85, 128, 134, and 163).

[32] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004. doi: 10.1016/S0196-6774(03)00097-X. (cit. on pp. 16, 39, 41, 63, 64, and 84).

[33] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algor.*, 3(2):19, 2007. doi: 10.1145/1240233.1240242. (cit. on p. 27).

[34] Amihood Amir, Mika Amit, Gad M. Landau, and Dina Sokol. Period recovery of strings over the Hamming and edit distances. *Theoretical Computer Science*, 710:2–18, 2018. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2017.10.026. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). (cit. on p. 23).

[35] Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *Proc. of ESA*, pages 5:1–5:18, 2019. doi: 10.4230/LIPIcs.ESA.2019.5. (cit. on p. 23).

[36] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12): 3707–3743, 2020. doi: 10.1007/S00453-020-00744-0. (cit. on p. 23).

[37] Amihood Amir, Ayelet Butman, Eitan Kondratovsky, Avivit Levy, and Dina Sokol. Multidimensional period recovery. *Algorithmica*, 84(6):1490–1510, 2022. doi: 10.1007/S00453-022-00926-Y. (cit. on p. 23).

[38] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *Proc. of FOCS*, pages 377–386, 2010. doi: 10.1109/FOCS.2010.43. (cit. on p. 25).

[39] Lorraine A.K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. ISSN 0167-8655. doi: https://doi.org/10.1016/j.patrec.2017.01.018. (cit. on pp. 17 and 26).

[40] Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proc. of SODA*, pages 572–591, 2015. doi: 10.1137/1.9781611973730.39. (cit. on p. 23).

[41] Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. *Theoretical Computer Science*, 494:13–23, 2013. (cit. on pp. 84 and 163).

[42] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proc. of STOC*, page 51–58, 2015. ISBN 9781450335362. doi: 10.1145/2746539.2746612. URL https://doi.org/10.1145/2746539.2746612. (cit. on p. 84).

[43] Arturs Backurs and Krzysztof Onak. Fast algorithms for parsing sequences of parentheses with few errors. In *Proc. of PODS 2016*, pages 477–488, 2016. doi: 10.1145/2902251.2902304. (cit. on pp. 84 and 127).

[44] Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theoretical Computer Science*, 922:271–282, 2022. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2022.04.029. (cit. on p. 23).

[45] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In *Proc. of SODA*, pages 562–571, 2015. doi: 10.1137/1.9781611973730.38. (cit. on p. 63).

[46] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52:3457–3467, 1995. doi: 10.1103/PhysRevA.52.3457. (cit. on p. 59).

[47] Gabriel Bathie and Tatiana Starikovskaya. Property testing of regular languages with applications to streaming property testing of visibly pushdown languages. In *Proc. of ICALP*, volume 198 of *LIPIcs*, pages 119:1–119:17, 2021. doi: 10.4230/LIPICS.ICALP.2021.119. URL https://doi.org/10.4230/LIPIcs.ICALP.2021.119. (cit. on p. 163).

[48] Gabriel Bathie and Tatiana Starikovskaya. Property Testing of Regular Languages with Applications to Streaming Property Testing of Visibly Pushdown Languages. In *Proc. of ICALP*, volume 198 of *LIPIcs*, pages 119:1–119:17, 2021. ISBN 978-3-95977-195-5. doi: 10.4230/LIPIcs.ICALP.2021.119. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2021.119. (cit. on pp. xv, 5, 83, 89, 90, 91, 94, 97, 101, 112, and 194).

[49] Gabriel Bathie, Tomasz Kociumaka, and Tatiana Starikovskaya. Small-Space Algorithms for the Online Language Distance Problem for Palindromes and Squares. In *Proc. of ISAAC*, pages 10:1–10:17, 2023. doi: 10.4230/LIPICS.ISAAC.2023.10. (cit. on pp. xv, 5, 23, 33, 40, 85, and 163).

[50] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Longest Common Extensions with Wildcards: Trade-off and Applications. In *Proc. of ESA*, 2024. (cit. on pp. xiv, 3, 22, and 43).

[51] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Internal Pattern Matching in Small Space and Applications. In *Proc. of CPM*, volume 296, pages 4:1–4:20, 2024. ISBN 978-3-95977-326-3. doi: 10.4230/LIPIcs.CPM.2024.4. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2024.4. (cit. on pp. xiii, 3, and 21).

[52] Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Pattern Matching with Mismatches and Wildcards. In *Proc. of ESA*, volume 308, pages 20:1–20:15, 2024. ISBN 978-3-95977-338-6. doi: 10.4230/LIPIcs.ESA.2024.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2024.20. (cit. on pp. xiv, 3, 22, 23, 32, 33, and 64).

[53] Gabriel Bathie, Jonas Ellert, and Tatiana Starikovskaya. Small space encoding and recognition of k-palindromic prefixes, 2025. URL https://arxiv.org/abs/2410.03309. (cit. on pp. xvi, 5, and 87).

[54] Tugkan Batu, Lance Fortnow, Ronitt Rubinfeld, Warren D Smith, and Patrick White. Testing that distributions are close. In *Proc. of FOCS*, pages 259–269. IEEE, 2000. (cit. on p. 81).

[55] Tugkan Batu, Eldar Fischer, Lance Fortnow, Ravi Kumar, Ronitt Rubinfeld, and Patrick White. Testing random variables for independence and identity. In *Proc. of FOCS*, pages 442–451. IEEE, 2001. (cit. on p. 81).

[56] Felix A. Behrend. On sets of integers which contain no three terms in arithmetical progression. *Proceedings of the National Academy of Sciences*, 32(12):331–332, 1946. (cit. on pp. 42 and 43).

[57] Djamal Belazzougui and Mathieu Raffinot. Approximate regular expression matching with multi-strings. *Journal of Discrete Algorithms*, 18:14–21, 2013. doi: 10.1016/j.jda.2012.07.008. (cit. on pp. 84 and 127).

[58] Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *Proc. of CPM*, pages 8:1–8:15, 2021. doi: 10.4230/LIPIcs.CPM.2021.8. (cit. on p. 27).

[59] Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In *Proc. of CPM*, pages 5:1–5:14, 2020. doi: 10.4230/LIPICS.CPM.2020.5. (cit. on p. 24).

[60] Petra Berenbrink, Funda Ergün, Frederik Mallmann-Trenn, and Erfan Sadeqi Azer. Palindrome recognition in the streaming model. In *Proc. of STACS*, volume 25, pages 149–161, 2014. doi: 10.4230/LIPIcs.STACS.2014.149. (cit. on pp. 128 and 162).

[61] Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching. In *Proc. of STOC*, pages 219–232, 2023. doi: 10.1145/3564246.3585239. (cit. on pp. xi, 1, 16, 85, 129, 134, and 135).

[62] Sudatta Bhattacharya and Michal Koucký. Locally consistent decomposition of strings with applications to edit distance sketching, 2023. URL `https://arxiv.org/abs/2302.04475`. (cit. on pp. 135, 136, 137, 138, 139, and 193).

[63] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind. String matching with variable length gaps. *Theoretical Computer Science*, 443:25–34, 2012. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2012.03.029. (cit. on pp. xii and 2).

[64] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. doi: 10.1016/J.JDA.2013.06.003. (cit. on pp. 32 and 63).

[65] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014. doi: 10.1007/S00224-013-9498-4. (cit. on p. 64).

[66] Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In *Proc. of CPM*, pages 65–76, 2015. doi: 10.1007/978-3-319-19929-0_6. (cit. on p. 63).

[67] Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, 2017. doi: 10.1016/J.JCSS.2017.01.002. (cit. on p. 63).

[68] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. Finger search in grammar-compressed strings. *Theory of Computing Systems*, 62: 1715–1735, 2018. doi: 10.1007/S00224-017-9839-9. (cit. on p. 63).

[69] Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proc. of SODA*, pages 607–626, 2020. doi: 10.1137/1.9781611975994.37. (cit. on pp. 12, 24, and 63).

[70] Francine Blanchet-Sadri and Justin Lazarow. Suffix trees for partial words and the longest common compatible prefix problem. In *Proc. of LATA 2013*, pages 165–176, 2013. ISBN 978-3-642-37064-9. doi: 10.1007/978-3-642-37064-9_16. (cit. on pp. 40 and 64).

[71] Francine Blanchet-Sadri and S. Osborne. Computing longest common extensions in partial words. *Discret. Appl. Math.*, 246:119–139, 2018. doi: 10.1016/J.DAM.2016.06.007. (cit. on p. 64).

[72] Francine Blanchet-Sadri, Rachel Harred, and Justin Lazarow. Longest common extensions in partial words. In *Proc. of IWOCA*, volume 9538 of *LNCS*, pages 52–64, 2015. doi: 10.1007/978-3-319-29516-9_5. URL https://doi.org/10.1007/978-3-319-29516-9_5. (cit. on p. 64).

[73] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. of STOC*, pages 73–83, 1990. (cit. on p. 81).

[74] Béla Bollobás and Shoham Letzter. Longest common extension. *Eur. J. Comb.*, 68: 242–248, 2018. doi: 10.1016/J.EJC.2017.07.019. (cit. on p. 64).

[75] A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. In *Proc. of STOC*, STOC '80, page 294–301, 1980. ISBN 0897910176. doi: 10.1145/800141.804677. URL https://doi.org/10.1145/800141.804677. (cit. on p. 162).

[76] Kirill Borozdin, Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palindromic Length in Linear Time. In *Proc. of CPM*, volume 78 of *LIPIcs*, pages 23:1–23:12, 2017. ISBN 978-3-95977-039-2. doi: 10.4230/LIPIcs.CPM.2017.23. URL http://drops.dagstuhl.de/opus/volltexte/2017/7338. (cit. on pp. xvi, 5, 86, 161, 162, 167, and 190).

[77] Dany Breslauer and Zvi Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4), 2014. ISSN 1549-6325. doi: 10.1145/2635814. URL https://doi.org/10.1145/2635814. (cit. on p. 15).

[78] Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real-time constant-space string matching. *Theoretical Computer Science*, 483:2–9, 2013. doi: 10.1016/J.TCS.2012.11.040. (cit. on pp. 24, 26, 184, and 187).

[79] Karl Bringmann, Fabrizio Grandoni, Barna Saha, and Virginia Vassilevska Williams. Truly subcubic algorithms for language edit distance and RNA folding via fast bounded-difference min-plus product. *SIAM Journal on Computing*, 48 (2):481–512, 2019. doi: 10.1137/17M112720X. (cit. on pp. 84 and 127).

[80] Karl Bringmann, Philip Wellnitz, and Marvin Künnemann. Few matches or almost periodicity: Faster pattern matching with mismatches in compressed texts. In *Proc. of SODA*, pages 1126–1145, 2019. doi: 10.1137/1.9781611975482.69. (cit. on pp. 39, 42, 46, and 48).

[81] Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, 2002. doi: 10.1016/S0304-3975(01)00144-X. (cit. on p. 59).

[82] Clément L Canonne. Are few bins enough: Testing histogram distributions. In *Proc. of PODS*, pages 455–463, 2016. (cit. on p. 81).

[83] Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. Information cost tradeoffs for augmented index and streaming language recognition. *SIAM J. Comput.*, 42(1):61–83, 2013. doi: 10.1137/100816481. URL https://doi.org/10.1137/100816481. (cit. on p. 34).

[84] Timothy M. Chan, Kasper Green Larsen, and Mihai Puatracscu. Orthogonal range searching on the RAM, revisited. In *Proc. of SoCG*, pages 1–10, 2011. doi: 10.1145/1998196.1998198. (cit. on p. 27).

[85] Timothy M. Chan, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Approximating text-to-pattern hamming distances. In *Proc. of STOC*, pages 643–656, 2020. doi: 10.1145/3357713.3384266. (cit. on pp. 16 and 39).

[86] Timothy M. Chan, Ce Jin, Virginia Vassilevska Williams, and Yinzhan Xu. Faster algorithms for text-to-pattern hamming distances. In *Proc. of FOCS*, pages 2188–2203, 2023. doi: 10.1109/FOCS57990.2023.00136. (cit. on pp. 16, 39, and 84).

[87] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *Proc. of CPM*, pages 8:1–8:15, 2020. doi: 10.4230/LIPICS.CPM.2020.8. (cit. on p. 23).

[88] Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Efficient enumeration of distinct factors using package representations. In *Proc. of SPIRE*, volume 12303, pages 247–261, 2020. doi: 10.1007/978-3-030-59212-7_18. (cit. on p. 23).

[89] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *Proc. of FOCS*, pages 978–989, 2020. doi: 10.1109/FOCS46700.2020.00095. (cit. on pp. xiii, 2, 3, 15, 16, 19, 20, 21, 23, 24, 32, 33, 39, 40, 41, 42, 43, 44, 47, 48, 51, 53, 57, 58, 63, 64, 129, 149, 150, 155, 156, 157, and 193).

[90] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi: 10.1007/S00453-021-00821-Y. (cit. on p. 23).

[91] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Faster algorithms for longest common substring. In *Proc. of*

ESA, pages 30:1–30:17, 2021. doi: 10.4230/LIPICS.ESA.2021.30. Full version: https://arxiv.org/abs/2105.03106. (cit. on p. 29).

[92] Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular pattern matching with k mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi: 10.1016/J.JCSS.2020.07.003. (cit. on pp. 17, 23, 44, and 46).

[93] Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate circular pattern matching. In *Proc. of ESA*, pages 35:1–35:19, 2022. doi: 10.4230/LIPICS.ESA.2022.35. (cit. on pp. 17, 23, 32, and 33).

[94] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *Proc. of FOCS*, pages 698–707, 2022. doi: 10.1109/FOCS54457.2022.00072. (cit. on pp. 16, 23, 32, and 33).

[95] Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest palindromic substring in sublinear time. In *Proc. of CPM*, volume 223, pages 20:1–20:9, 2022. doi: 10.4230/LIPICS.CPM.2022.20. (cit. on p. 63).

[96] Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching under edit distance. In *Proc. of STACS*, pages 24:1–24:22, 2024. doi: 10.4230/LIPIcs.STACS.2024.24. (cit. on pp. 17, 32, 33, and 59).

[97] Kuei-Hao Chen, Guan-Shieng Huang, and Richard Chia-Tung Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *The Computer Journal*, 57(5): 731–743, 03 2013. ISSN 0010-4620. doi: 10.1093/comjnl/bxt023. (cit. on pp. 17 and 26).

[98] Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. Faster min-plus product for monotone instances. In *Proc. of STOC*, pages 1529–1542, 2022. doi: 10.1145/3519935.3520057. (cit. on pp. 84 and 127).

[99] Noam Chomsky and Marcel-Paul Schützenberger. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems*, volume 26, pages 118–161. 1959. doi: https://doi.org/10.1016/S0049-237X(09)70104-1. URL https://www.sciencedirect.com/science/article/pii/S0049237X09701041. (cit. on p. 194).

[100] Peter Clifford and Raphaël Clifford. Simple deterministic wildcard matching. *Inf. Process. Lett.*, 101(2):53–54, 2007. doi: 10.1016/j.ipl.2006.08.002. (cit. on pp. 17, 18, 22, 40, 42, 64, 70, and 77).

[101] Raphaël Clifford and Ely Porat. A filtering algorithm for k-mismatch with don't cares. *Inf. Process. Lett.*, 110(22):1021–1025, 2010. doi: 10.1016/j.ipl.2010.08.012. (cit. on pp. 40 and 41).

[102] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. From coding theory to efficient pattern matching. In *Proc. of SODA*, pages 778–784. SIAM, 2009. (cit. on pp. 17, 41, and 64).

[103] Raphaël Clifford, Klim Efremenko, Ely Porat, and Amir Rothschild. Pattern matching with don't cares and few errors. *J. Comput. Syst. Sci.*, 76(2):115–124, 2010. doi: 10.1016/j.jcss.2009.06.002. (cit. on pp. 41 and 64).

[104] Raphaël Clifford, Klim Efremenko, Benny Porat, and Ely Porat. A black box for online approximate pattern matching. *Inf. Comput.*, 209(4):731–736, 2011. doi: 10.1016/J.IC.2010.12.007. (cit. on p. 36).

[105] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. The k-mismatch problem revisited. In *Proc. of SODA*, pages 2039–2052, 2016. doi: 10.1137/1.9781611974331.CH142. (cit. on pp. 16 and 39).

[106] Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *Proc. of STACS*, pages 22:1–22:14, 2018. doi: 10.4230/LIPICS.STACS.2018.22. (cit. on p. 64).

[107] Raphaël Clifford, Tomasz Kociumaka, and Ely Porat. The streaming k-mismatch problem. In *Proc. of SODA*, pages 1106–1125, 2019. doi: 10.1137/1.9781611975482.68. (cit. on pp. xi, 1, 16, 85, 129, 134, 142, 143, 145, 150, and 193).

[108] John Cocke. *Programming languages and their compilers: Preliminary notes*. New York University, 1969. (cit. on pp. xv and 4).

[109] Vincent Cohen-Addad, CS Karthik, and Guillaume Lagarde. On efficient low distortion ultrametric embedding. In *Proc. of ICML*, pages 2078–2088. PMLR, 2020. (cit. on p. 199).

[110] Vincent Cohen-Addad, Rémi De Joannis De Verclos, and Guillaume Lagarde. Improving ultrametrics embeddings through coresets. In *Proc. of ICML*, pages 2060–2068. PMLR, 2021. (cit. on p. 199).

[111] Richard Cole and Ramesh Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. of STOC*, pages 592–601, 2002. doi: 10.1145/509907.509992. (cit. on pp. 17, 40, 42, and 64).

[112] Richard Cole and Ramesh Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002. doi: 10.1137/S0097539700370527. URL https://doi.org/10.1137/S0097539700370527. (cit. on p. 16).

[113] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. of STOC*, pages 91–100, 2004. doi: 10.1145/1007352.1007374. (cit. on p. 64).

[114] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002. ISBN 978-981-02-4782-9. doi: 10.1142/4838. (cit. on p. 66).

[115] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. ISBN 978-0-521-84899-2. doi: 10.1017/cbo9780511546853. (cit. on pp. 8 and 66).

[116] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Walen. A note on the longest common compatible prefix problem for partial words. *J. Discrete Algorithms*, 34:49–53, 2015. doi: 10.1016/J.JDA.2015.05.003. (cit. on p. 40).

[117] Maxime Crochemore, Costas S Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Waleń. A note on the longest common compatible prefix problem for partial words. *Journal of Discrete Algorithms*, 34:49–53, 2015. doi: 10.1016/J.JDA.2015.05.003. (cit. on pp. 22, 64, 65, 68, 70, 193, and 194).

[118] Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Internal quasiperiod queries. In *Proc. of SPIRE*, pages 60–75, 2020. doi: 10.1007/978-3-030-59212-7_5. (cit. on p. 23).

[119] Jiangqi Dai, Qingyu Shi, and Tingqiang Xu. Faster algorithms for internal dictionary queries. *CoRR*, abs/2312.11873, 2023. doi: 10.48550/ARXIV.2312.11873. (cit. on p. 23).

[120] Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved approximation algorithms for Dyck edit distance and RNA folding. In *Proc. of ICALP*, volume 229 of *LIPIcs*, pages 49:1–49:20, 2022. doi: 10.4230/LIPIcs.ICALP.2022.49. (cit. on pp. 23, 84, and 127).

[121] Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, and Kaiyu Wu. Internal masked prefix sums and its connection to fully internal measurement queries. In *Proc. of SPIRE*, pages 217–232, 2022. ISBN 978-3-031-20643-6. doi: 10.1007/978-3-031-20643-6_16. (cit. on p. 23).

[122] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of SOSP*, 2007. (cit. on p. 82).

[123] Bartlomiej Dudek, Pawel Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya. Streaming regular expression membership and pattern matching. In *Proc. of SODA*, pages 670–694, 2022. doi: 10.1137/1.9781611977073.30. URL https://doi.org/10.1137/1.9781611977073.30. (cit. on p. 163).

[124] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4 (4):363–381, 1983. doi: 10.1016/0196-6774(83)90017-2. (cit. on p. 26).

[125] Anouk Duyster and Tomasz Kociumaka. Logarithmic-time internal pattern matching queries in compressed and dynamic texts. In *SPIRE*, pages 102–117, 2024. doi: 10.1007/978-3-031-72200-4_8. (cit. on pp. 21 and 58).

[126] Anita Dürr. Improved bounds for rectangular monotone Min-Plus Product and applications. *Information Processing Letters*, 181:106358, 2023. ISSN 0020-0190. doi: https://doi.org/10.1016/j.ipl.2023.106358. (cit. on p. 127).

[127] Michael Elkin. An improved construction of progression-free sets. In *Proc. of SODA*, pages 886–905, 2010. doi: 10.1137/1.9781611973075.72. (cit. on pp. xiv, 3, 22, 42, 43, and 59).

[128] Mourad Elloumi. Algorithms for next-generation sequencing data. *Techniques, approaches, and applications*, 2017. (cit. on pp. xi and 1).

[129] Paul Erdös and Paul Turán. On some sequences of integers. *Journal of the London Mathematical Society*, s1-11(4):261–264, 1936. doi: https://doi.org/10.1112/jlms/s1-11.4.261. (cit. on pp. 42 and 43).

[130] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. of FOCS*, pages 137–143. IEEE, 1997. (cit. on p. 12).

[131] Martin Farach, Sampath Kannan, and Tandy Warnow. A robust model for finding optimal evolutionary trees. In *Proc. of STOC*, pages 137–145, 1993. (cit. on p. 199).

[132] Henning Fernau, Florin Manea, Robert Mercaş, and Markus L. Schmid. Pattern matching with variables: Efficient algorithms and complexity results. *ACM Trans. Comput. Theory*, 12(1), feb 2020. ISSN 1942-3454. doi: 10.1145/3369935. URL https://doi.org/10.1145/3369935. (cit. on p. 23).

[133] Gabriele Fici, Travis Gagie, Juha Kärkkäinen, and Dominik Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014. doi: 10.1016/J.JDA.2014.08.001. URL https://doi.org/10.1016/j.jda.2014.08.001. (cit. on pp. 86 and 161).

[134] Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. ISSN 00029939. doi: 10.1090/S0002-9939-1965-0174934-9. (cit. on pp. 8, 18, 148, and 149).

[135] Eldar Fischer, Guy Kindler, Dana Ron, Shmuel Safra, and Alex Samorodnitsky. Testing juntas. *Journal of Computer and System Sciences*, 68(4):753–787, 2004. (cit. on p. 81).

[136] Eldar Fischer, Frédéric Magniez, and Tatiana Starikovskaya. Improved bounds for testing Dyck languages. In *Proc. of SODA*, pages 1529–1544. SIAM, 2018. (cit. on pp. 81, 83, 92, and 194).

[137] Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. of CPM*, pages 160–171, 2015. doi: 10.1007/978-3-319-19929-0_14. (cit. on p. 26).

[138] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. of CPM*, pages 36–48, 2006. ISBN 978-3-540-35461-1. (cit. on pp. 63, 69, and 77).

[139] Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. of ESA*, volume 9294, pages 533–544, 2015. doi: 10.1007/978-3-662-48350-3_45. (cit. on p. 26).

[140] Michael J. Fischer and Michael S. Paterson. String-matching and other products. In *SCC*, pages 113–125, 1974. (cit. on pp. 17, 40, and 64).

[141] Nick Fischer. Deterministic sparse pattern matching via the Baur-Strassen theorem. In *Proc. of SODA*, pages 3333–3353, 2024. doi: 10.1137/1.9781611977912.119. (cit. on p. 64).

[142] Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre. Streaming property testing of visibly pushdown languages. In *Proc. of ESA*, volume 57 of *LIPIcs*, pages 43:1–43:17, 2016. ISBN 978-3-95977-015-6. doi: 10.4230/LIPIcs.ESA.2016.43. (cit. on pp. 81, 82, 83, 92, 94, 163, and 194).

[143] Kimmo Fredriksson and Szymon Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009. ISSN 1570-8667. doi: https://doi.org/10.1016/j.jda.2008.09.001. (cit. on pp. 17 and 26).

[144] Dvir Fried, Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, Ely Porat, and Tatiana Starikovskaya. An improved algorithm for the k-Dyck edit distance problem. In *Proc. of SODA*, pages 3650–3669, 2022. doi: 10.1137/1.9781611977073.144. (cit. on p. 127).

[145] Zvi Galil. On converting on-line algorithms into real-time and on real-time algorithms for string-matching and palindrome recognition. *SIGACT News*, 7(4): 26–30, nov 1975. ISSN 0163-5700. doi: 10.1145/990502.990505. URL https://doi.org/10.1145/990502.990505. (cit. on p. 86).

[146] Zvi Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proc. of STOC*, pages 161–173, 1976. doi: 10.1145/800113.803644. (cit. on p. 128).

[147] Zvi Galil and Raffaele Giancarlo. Improved string matching with k mismatches. *ACM SIGACT News*, 17(4):52–54, 1986. (cit. on pp. 63 and 128).

[148] Zvi Galil and Joel Seiferas. A Linear-Time On-Line Recognition Algorithm for "Palstar". *J. ACM*, 25(1):102–111, jan 1978. ISSN 0004-5411. doi: 10.1145/322047.322056. URL https://doi.org/10.1145/322047.322056. (cit. on pp. 86 and 161).

[149] Moses Ganardi. Visibly pushdown languages over sliding windows. In *Proc. of STACS*, volume 126 of *LIPIcs*, pages 29:1–29:17, 2019. ISBN 978-3-95977-100-9. doi: 10.4230/LIPIcs.STACS.2019.29. (cit. on pp. 84 and 163).

[150] Moses Ganardi, Danny Hucke, and Markus Lohrey. Querying regular languages over sliding windows. In *Proc. of FSTTCS*, volume 65 of *LIPIcs*, pages 18:1–18:14, 2016. doi: 10.4230/LIPIcs.FSTTCS.2016.18. (cit. on p. 163).

[151] Moses Ganardi, Danny Hucke, Daniel König, Markus Lohrey, and Konstantinos Mamouras. Automata theory on sliding windows. In *Proc. of STACS*, volume 96 of *LIPIcs*, pages 31:1–31:14, 2018. doi: 10.4230/LIPIcs.STACS.2018.31. (cit. on p. 163).

[152] Moses Ganardi, Danny Hucke, and Markus Lohrey. Randomized sliding window algorithms for regular languages. In *Proc. of ICALP*, volume 107 of *LIPIcs*, pages 127:1–127:13, 2018. doi: 10.4230/LIPIcs.ICALP.2018.127. (cit. on pp. 90 and 163).

[153] Moses Ganardi, Artur Jeż, and Markus Lohrey. Sliding windows over context-free languages. In *Proc. of MFCS*, volume 117 of *LIPIcs*, pages 15:1–15:15, 2018. doi: 10.4230/LIPIcs.MFCS.2018.15. (cit. on pp. 84 and 163).

[154] Moses Ganardi, Danny Hucke, Markus Lohrey, and Tatiana Starikovskaya. Sliding Window Property Testing for Regular Languages. In *Proc. of ISAAC*, volume 149 of *LIPIcs*, pages 6:1–6:13, 2019. ISBN 978-3-95977-130-6. doi: 10.4230/LIPIcs.ISAAC.2019.6. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ISAAC.2019.6. (cit. on pp. 92 and 163).

[155] Moses Ganardi, Louis Jachiet, Markus Lohrey, and Thomas Schwentick. Low-Latency Sliding Window Algorithms for Formal Languages. In *Proc. of FSTTCS*, volume 250 of *LIPIcs*, pages 38:1–38:23, 2022. ISBN 978-3-95977-261-7. doi: 10.4230/LIPIcs.FSTTCS.2022.38. (cit. on p. 84).

[156] Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha. How compression and approximation affect efficiency in string distance measures. In *Proc. of SODA*, pages 2867–2919, 2022. doi: 10.1137/1.9781611977073.112. (cit. on pp. 140 and 148).

[157] Pawel Gawrychowski and Tomasz Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. of SODA*, pages 425–439, 2017. doi: 10.1137/1.9781611974782.27. (cit. on pp. 12 and 63).

[158] Pawel Gawrychowski and Przemyslaw Uznanski. Towards unified approximate pattern matching for Hamming and L_1 distance. In *Proc. of ICALP*, volume 107 of *LIPIcs*, pages 62:1–62:13, 2018. doi: 10.4230/LIPICS.ICALP.2018.62. (cit. on pp. 16, 39, and 73).

[159] Pawel Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018. doi: 10.1007/S00224-017-9794-5. (cit. on p. 23).

[160] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. In *Proc. of SODA*, pages 1509–1528. SIAM, 2018. doi: 10.1137/1.9781611975031.99. (cit. on pp. 21, 58, and 63).

[161] Pawel Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemyslaw Uznanski. Tight tradeoffs for real-time approximation of longest palindromes in streams. *Algorithmica*, 81(9):3630–3654, 2019. doi: 10.1007/s00453-019-00591-8. (cit. on pp. 85, 128, 140, 141, and 163).

[162] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. Supporting descendants in simd-accelerated jsonpath. In *Proc. of ASPLOS*, volume 4, pages 338–361, 2023. (cit. on pp. xii and 2).

[163] Shay Golan, Tsvi Kopelowitz, and Ely Porat. Streaming pattern matching with d wildcards. *Algorithmica*, 81(5):1988–2015, 2019. doi: 10.1007/S00453-018-0521-7. (cit. on pp. 17 and 64).

[164] Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. The Streaming k-Mismatch Problem: Tradeoffs Between Space and Total Time. In *Proc. of CPM*, volume 161 of *LIPIcs*, pages 15:1–15:15, 2020. ISBN 978-3-95977-149-8. doi: 10.4230/LIPIcs.CPM.2020.15. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2020.15. (cit. on p. 16).

[165] Oded Goldreich. *Introduction to property testing*. Cambridge University Press, 2017. doi: 10.1017/9781108135252. (cit. on p. 81).

[166] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, jul 1998. ISSN 0004-5411. doi: 10.1145/285055.285060. URL https://doi.org/10.1145/285055.285060. (cit. on pp. 81 and 89).

[167] Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming for aibohphobes: Longest palindrome with mismatches. In *Proc. of FSTTCS*, volume 93, pages 31:1–31:13, 2018. ISBN 978-3-95977-055-2. doi: 10.4230/LIPIcs.FSTTCS.2017.31. (cit. on p. 128).

[168] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 0-521-58519-8. doi: 10.1017/cbo9780511574931. URL https://doi.org/10.1017/cbo9780511574931. (cit. on pp. 11, 12, and 63).

[169] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978. ISSN 0098-3500. doi: 10.1145/355791.355796. (cit. on pp. 65 and 66).

[170] Yijie Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *J. Algorithms*, 50(1):96–105, 2004. doi: 10.1016/j.jalgor.2003.09.001. (cit. on pp. 48 and 54).

[171] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi: 10.1137/0213024. (cit. on p. 63).

[172] Ramesh Hariharan and V. Vinay. String matching in Õ(sqrt(n)+sqrt(m)) quantum time. *Journal of Discrete Algorithms*, 1(1):103–110, 2003. doi: 10.1016/S1570-8667(03)00010-8. (cit. on p. 59).

[173] Johan Håstad. The shrinkage exponent of de morgan formulas is 2. *SIAM Journal on Computing*, 27(1):48–64, 1998. Preliminary version in FOCS'93. (cit. on p. 199).

[174] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *The collected works of Wassily Hoeffding*, pages 409–426, 1994. (cit. on p. 105).

[175] Jan Holub and William F. Smyth. Algorithms on indeterminate strings. In *Proc. of IWOCA*, page 36–45, 2003. (cit. on p. 67).

[176] Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct indexes for circular patterns. In *Proc. of ISAAC*, pages 673–682, 2011. ISBN 978-3-642-25591-5. doi: 10.1007/978-3-642-25591-5_69. (cit. on pp. 17 and 26).

[177] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, and Sharma V. Thankachan. Space-efficient construction algorithm for the circular suffix tree. In *Proc. of CPM*, pages 142–152, 2013. ISBN 978-3-642-38905-4. doi: 10.1007/978-3-642-38905-4_15. (cit. on pp. 17 and 26).

[178] Wei Huang, Yaoyun Shi, Shengyu Zhang, and Yufan Zhu. The communication complexity of the Hamming distance problem. *Information Processing Letters*, 99 (4):149–153, 2006. (cit. on p. 135).

[179] Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. of CPM*, volume 8486 of *LNCS*, pages 150–161, 2014. doi: 10.1007/978-3-319-07566-2_16. URL https://doi.org/10.1007/978-3-319-07566-2_16. (cit. on pp. 86 and 161).

[180] Costas S. Iliopoulos and Jakub Radoszewski. Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties. In *Proc. of CPM*, volume 54, pages 8:1–8:12, 2016. ISBN 978-3-95977-012-5. doi: 10.4230/LIPIcs.CPM.2016.8. (cit. on pp. 22, 64, 65, 67, 68, and 76).

[181] Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. In *Proc. of PSC 2002*, pages 65–74, 2002. (cit. on p. 67).

[182] Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and indexing circular patterns. In *Algorithms for Next-Generation Sequencing Data: Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. ISBN 978-3-319-59826-0. doi: 10.1007/978-3-319-59826-0_3. (cit. on pp. 17 and 26).

[183] Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Linear-time computation of cyclic roots and cyclic covers of a string. In *Proc. of CPM*, pages 15:1–15:15, 2023. doi: 10.4230/LIPICS.CPM.2023.15. (cit. on p. 23).

[184] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Proc. of FOCS*, pages 166–173, 1998. doi: 10.1109/SFCS.1998.743440. (cit. on pp. 17, 40, and 64).

[185] Rahul Jain and Ashwin Nayak. The space complexity of recognizing well-parenthesized expressions in the streaming model: The index function revisited. *IEEE Transactions on Information Theory*, 60(10):6646–6668, Oct 2014. doi: 10.1109/TIT.2014.2339859. (cit. on p. 163).

[186] Ce Jin and Jakob Nogler. Quantum speed-ups for string synchronizing sets, longest common substring, and k-mismatch matching. In *Proc. of SODA*, pages 5090–5121, 2023. doi: 10.1137/1.9781611977554.ch186. (cit. on pp. 40 and 59).

[187] Ce Jin and Yinzhan Xu. Shaving logs via large sieve inequality: Faster algorithms for sparse convolution and more. In *Proc. of STOC*, pages 1573–1584, 2024. doi: 10.1145/3618260.3649605. (cit. on pp. 16 and 39).

[188] jq Contributors. jq, is a lightweight and flexible command-line JSON processor. https://jqlang.org/, 2025. Accessed: 2025-02-13. (cit. on pp. xii and 2).

[189] Adam Kalai. Efficient pattern-matching with don't cares. In *Proc. of SODA*, pages 655–656, 2002. (cit. on pp. 17, 40, and 64).

[190] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. (cit. on pp. 15, 16, and 134).

[191] Marek Karpinski and Yakov Nekrich. Space efficient multi-dimensional range reporting. In *Proc. of COCOON*, volume 5609, pages 215–224, 2009. doi: 10.1007/978-3-642-02882-3_22. (cit. on p. 27).

[192] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*, 1966. (cit. on pp. xv and 4).

[193] Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi: 10.1016/J.TCS.2013.10.010. (cit. on p. 23).

[194] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proc. of STOC*, pages 756–767, 2019. doi: 10.1145/3313276.3316368. (cit. on pp. 57 and 63).

[195] Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *Proc. of FOCS*, pages 1002–1013, 2020. doi: 10.1109/FOCS46700.2020.00097. (cit. on p. 24).

[196] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *Proc. of STOC*, pages 1657–1670, 2022. doi: 10.1145/3519935.3520061. URL https://doi.org/10.1145/3519935.3520061. Full version at http://arxiv.org/abs/1910.10631. (cit. on pp. 24 and 59).

[197] Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *Proc. of FOCS*, pages 1877–1886, 2023. doi: 10.1109/FOCS57990.2023.00114. (cit. on p. 63).

[198] Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-end parsing. In *Proc. of SODA*, pages 2847–2866, 2022. doi: 10.1137/1.9781611977073.111. (cit. on p. 63).

[199] Stephen C. Kleen. Representationof events in nerve nets and finite automata. *RAND Corporation*, 1951. (cit. on pp. xi and 1).

[200] Donald E Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8 (2):18–24, 1976. (cit. on p. 9).

[201] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi: 10.1137/0206024. URL https://doi.org/10.1137/0206024. (cit. on pp. xi, 1, 15, 39, and 86).

[202] Tomasz Kociumaka. *Efficient data structures for internal queries in texts*. PhD thesis, University of Warsaw, October 2018. Available at https://depotuw.ceon.pl/handle/item/3614. (cit. on pp. 23, 32, and 63).

[203] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Optimal data structure for internal pattern matching queries in a text and applications. *CoRR*, abs/1311.6235, 2013. (cit. on pp. 23, 24, and 32).

[204] Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *Proc. of ESA*, pages 605–617, 2014. doi: 10.1007/978-3-662-44777-2_50. (cit. on pp. 25 and 162).

[205] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proc. of SODA*, pages 532–551, 2015. doi: 10.1137/1.9781611973730.36. (cit. on pp. 23 and 32).

[206] Tomasz Kociumaka, Ritu Kundu, Manal Mohamed, and Solon P. Pissis. Longest unbordered factor in quasilinear time. In *Proc. of ISAAC*, pages 70:1–70:13, 2018. doi: 10.4230/LIPIcs.ISAAC.2018.70. (cit. on p. 23).

[207] Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya. Small-space and streaming pattern matching with k edits. In *Proc. of FOCS*, pages 885–896, 2021. doi: 10.1109/FOCS52979.2021.00090. (cit. on pp. 16, 131, 149, 155, and 156).

[208] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53 (5):1524–1577, 2024. doi: 10.1137/23M1567618. (cit. on pp. 46, 53, and 57).

[209] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. of FOCS*, pages 596–604, 1999. (cit. on p. 63).

[210] Roman Kolpakov and Gregory Kucherov. Finding approximate repetitions under Hamming distance. *Theoretical Computer Science*, 303(1):135–156, 2003. ISSN 0304-3975. doi: https://doi.org/10.1016/S0304-3975(02)00448-6. Logic and Complexity in Computer Science. (cit. on p. 128).

[211] Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009. (cit. on p. 63).

[212] Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. *Journal of Discrete Algorithms*, 46-47:1–15, 2017. ISSN 1570-8667. doi: https://doi.org/10.1016/j.jda.2017.10.004. (cit. on p. 23).

[213] S. Rao Kosaraju. Efficient string matching. Unpublished manuscript, 1987. (cit. on pp. 16, 39, and 41).

[214] Dmitry Kosolobov. Tight lower bounds for the longest common extension problem. *Information Processing Letters*, 125:26–29, 2017. doi: 10.1016/J.IPL.2017.05.003. (cit. on p. 63).

[215] Dmitry Kosolobov and Nikita Sivukhin. Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space. In *Proc. of CPM*, volume 296, pages 20:1–20:18, 2024. ISBN 978-3-95977-326-3. doi: 10.4230/LIPIcs.CPM.2024.20. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2024.20. (cit. on pp. xiii, 3, 12, 21, 24, 26, 28, 29, 32, 63, and 65).

[216] Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur. Palkis linear recognizable online. In *SOFSEM 2015: Theory and Practice of Computer Science*, pages 289–301, 2015. ISBN 978-3-662-46078-8. (cit. on pp. 86 and 161).

[217] Dmitry Kosolobov, Florin Manea, and Dirk Nowotka. Detecting one-variable patterns. In *Proc. of SPIRE*, pages 254–270, 2017. doi: 10.1007/978-3-319-67428-5_22. (cit. on p. 23).

[218] Michal Koucký and Michael E. Saks. Simple, deterministic, fast (but weak) approximations to edit distance and Dyck edit distance. In *Proc. of SODA*, pages 5203–5219, 2023. doi: 10.1137/1.9781611977554.ch188. (cit. on p. 127).

[219] Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *Proc. of MFCS*, volume 6907 of *LNCS*, pages 412–423, 2011. ISBN 978-3-642-22993-0. doi: 10.1007/978-3-642-22993-0_38. (cit. on pp. 127 and 163).

[220] Marvin Künnemann. On nondeterministic derandomization of Freivalds' algorithm: Consequences, avenues and algorithmic progress. In *Proc. of ESA*, volume 112 of *LIPIcs*, pages 56:1–56:16, 2018. ISBN 978-3-95977-081-1. doi: 10.4230/LIPIcs.ESA.2018.56. (cit. on pp. 65 and 66).

[221] Konstantin Kutzkov. Deterministic algorithms for skewed matrix products. In *Proc. of STACS*, volume 20 of *LIPIcs*, pages 466–477, 2013. ISBN 978-3-939897-50-7. doi: 10.4230/LIPIcs.STACS.2013.466. (cit. on pp. 65 and 66).

[222] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, volume 1. BG Teubner, 1909. (cit. on p. 9).

[223] Gad M. Landau and Jeanette P. Schmidt. An algorithm for approximate tandem repeats. In *Proc. of CPM*, pages 120–133, 1993. doi: 10.1007/BFb0029801. (cit. on p. 128).

[224] Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proc. of STOC*, page 220–230, 1986. ISBN 0897911938. doi: 10.1145/12130.12152. URL https://doi.org/10.1145/12130.12152. (cit. on pp. 22, 44, 63, 64, 65, 66, 193, and 194).

[225] Gad M Landau and Uzi Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. (cit. on p. 63).

[226] Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi: 10.1016/0022-0000(88)90045-1. (cit. on pp. 20, 21, 84, and 156).

[227] Gad M. Landau and Uzi Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989. doi: 10.1016/0196-6774(89)90010-2. (cit. on p. 43).

[228] Lillian Lee. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15, jan 2002. ISSN 0004-5411. doi: 10.1145/505241.505242. (cit. on pp. 84 and 127).

[229] Moshe Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302, 2013. doi: 10.1007/978-3-642-40273-9_18. (cit. on p. 24).

[230] M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005. (cit. on pp. 17 and 26).

[231] Frédéric Magniez and Michel de Rougemont. Property testing of regular tree languages. *Algorithmica*, 49(2):127–146, 2007. doi: 10.1007/s00453-007-9028-3. (cit. on pp. 81, 82, 83, 92, and 94).

[232] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM J. Comput.*, 43(6):1880–1905, 2014. doi: 10.1137/130926122. (cit. on p. 163).

[233] Tung Mai, Anup Rao, Ryan A Rossi, and Saeed Seddighin. Optimal space and time for streaming pattern matching. *arXiv preprint arXiv:2107.04660*, 2021. (cit. on p. 25).

[234] Glenn Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, jul 1975. ISSN 0004-5411. doi: 10.1145/321892.321896. URL https://doi.org/10.1145/321892.321896. (cit. on p. 86).

[235] Udi Manber and Ricardo A. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Inf. Process. Lett.*, 37(3):133–136, 1991. doi: 10.1016/0020-0190(91)90032-D. (cit. on p. 40).

[236] Florin Manea, Robert Mercas, and Catalin Tiseanu. An algorithmic toolbox for periodic partial words. *Discret. Appl. Math.*, 179:174–192, 2014. doi: 10.1016/J.DAM.2014.07.017. (cit. on p. 64).

[237] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980. (cit. on p. 84).

[238] Gerhard Mehldau and Gene Myers. A system for pattern matching applications on biosequences. *Bioinformatics*, 9(3):299–314, 1993. doi: 10.1093/bioinformatics/9.3.299. (cit. on p. 40).

[239] Meilisearch. Meilisearch search engine. https://github.com/meilisearch/meilisearch, 2025. Accessed: 2025-04-11. (cit. on p. 195).

[240] Kazuki Mitani, Takuya Mieno, Kazuhisa Seto, and Takashi Horiyama. Internal longest palindrome queries in optimal time. In *Proc. of WALCOM*, pages 127–138, 2023. (cit. on p. 23).

[241] Michael Mitzenmacher and Salil Vadhan. Lecture 6: RAM model. https://people.seas.harvard.edu/ cs125/fall14/lec6.pdf, 2014. Accessed: 2025-04-03. (cit. on p. 9).

[242] Gene Myers. Approximately matching context-free languages. *Information Processing Letters*, 54(2):85–92, 1995. ISSN 0020-0190. doi: 10.1016/0020-0190(95)00007-y. (cit. on pp. 84 and 127).

[243] Yinzhan Xu Nick Fischer, Ce Jin. New applications of 3SUM-counting in fine-grained complexity and pattern matching. In *Proc. of SODA*, 2025. (cit. on p. 41).

[244] Marius Nicolae and Sanguthevar Rajasekaran. On string matching with mismatches. *Algorithms*, 8(2):248–270, 2015. doi: 10.3390/a8020248. (cit. on pp. 41 and 64).

[245] Marius Nicolae and Sanguthevar Rajasekaran. On pattern matching with k mismatches and few don't cares. *Inf. Process. Lett.*, 118:78–82, 2017. doi: 10.1016/j.ipl.2016.10.003. (cit. on pp. 40, 41, and 64).

[246] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. of MFCS*, volume 58 of *LIPIcs*, pages 72:1–72:15, 2016. doi: 10.4230/LIPIcs.MFCS.2016.72. (cit. on pp. 63 and 135).

[247] Michal Parnas, Dana Ron, and Alex Samorodnitsky. Testing basic boolean formulae. *SIAM Journal on Discrete Mathematics*, 16(1):20–46, 2002. (cit. on p. 81).

[248] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. Testing membership in parenthesis languages. *Random Struct. Algorithms*, 22(1):98–138, 2003. doi: 10.1002/rsa.10067. (cit. on pp. 81, 83, and 92).

[249] Jean-Éric Pin, editor. *Handbook of Automata Theory*. EMS Press, sep 2021. ISBN 9783985475025. doi: 10.4171/automata-1. URL http://dx.doi.org/10.4171/AUTOMATA-1. (cit. on pp. 92, 93, 102, and 103).

[250] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proc. of FOCS*, pages 315–323. IEEE, 2009. (cit. on pp. 15, 16, and 19).

[251] Alexandre H. L. Porto and Valmir Carneiro Barbosa. Finding approximate palindromes in strings. *Pattern Recognit.*, 35(11):2581–2591, 2002. doi: 10.1016/S0031-3203(01)00179-0. (cit. on p. 128).

[252] Nicola Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Trans. Algorithms*, 17(1), dec 2021. ISSN 1549-6325. doi: 10.1145/3426870. URL https://doi.org/10.1145/3426870. (cit. on p. 63).

[253] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi: 10.1137/09075336X. (cit. on p. 64).

[254] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959. (cit. on p. 83).

[255] Jakub Radoszewski and Tatiana Starikovskaya. Streaming k-mismatch with error correcting and applications. *Information and Computation*, 271:104513, 2020. (cit. on p. 16).

[256] Mikhail Rubinchik and Arseny M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018. doi: 10.1016/J.EJC.2017.07.021. URL https://doi.org/10.1016/j.ejc.2017.07.021. (cit. on pp. 86 and 161).

[257] Mikhail Rubinchik and Arseny M. Shur. Palindromic k-factorization in pure linear time. In *Proc. of MFCS*, volume 170 of *LIPIcs*, pages 81:1–81:14, 2020. doi: 10.4230/LIPICS.MFCS.2020.81. URL https://doi.org/10.4230/LIPIcs.MFCS.2020.81. (cit. on pp. 86, 161, and 162).

[258] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM J. Comp.*, 25(2):252–271, 1996. doi: 10.1137/S0097539793255151. (cit. on p. 81).

[259] Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Proc. of ICALP*, volume 5125, pages 84–95, 2008. doi: 10.1007/978-3-540-70575-8_8. (cit. on p. 26).

[260] Imre Z Ruzsa and Endre Szemerédi. Triple systems with no six points carrying three triangles. *Combinatorics (Keszthely, 1976), Coll. Math. Soc. J. Bolyai*, 18 (939-945):2, 1978. (cit. on p. 82).

[261] Walter L. Ruzzo. On the complexity of general context-free language parsing and recognition. In *Proc. of ICALP*, volume 71 of *LNCS*, pages 489–497, 1979. doi: 10.1007/3-540-09510-1_39. (cit. on pp. 84 and 127).

[262] Wojciech Rytter. On maximal suffixes and constant-space linear-time versions of KMP algorithm. *Theoretical Computer Science*, 299(1-3):763–774, 2003. doi: 10.1016/S0304-3975(02)00590-X. (cit. on pp. 149 and 155).

[263] Barna Saha. The Dyck language edit distance problem in near-linear time. In *Proc. of FOCS*, pages 611–620, 2014. doi: 10.1109/FOCS.2014.71. (cit. on p. 127).

[264] Barna Saha. Language edit distance and maximum likelihood parsing of stochastic grammars: Faster algorithms and connection to fundamental graph problems. In *Proc. of FOCS*, pages 118–135, 2015. doi: 10.1109/FOCS.2015.17. (cit. on p. 127).

[265] Barna Saha. Fast space-efficient approximations of language edit distance and RNA folding: An amnesic dynamic programming approach. In *Proc. of FOCS*, pages 295–306, 2017. doi: 10.1109/FOCS.2017.35. (cit. on p. 127).

[266] Michael Saks and C. Seshadhri. Space efficient streaming algorithms for the distance to monotonicity and asymmetric edit distance. In *Proc. of SODA*, pages 1698–1709, 2013. doi: 10.1137/1.9781611973105.122. (cit. on p. 25).

[267] Giorgio Satta. Tree-adjoining grammar parsing and boolean matrix multiplication. *Comput. Linguistics*, 20(2):173–191, 1994. URL https://aclanthology.org/J94-2002. (cit. on p. 127).

[268] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(70)80006-X. (cit. on p. 123).

[269] Philipp Schepper. Fine-grained complexity of regular expression pattern matching and membership. In *Proc. of ESA*, 2020. (cit. on pp. xiv and 4).

[270] Anatol O. Slisenko. A simplified proof of the real-time recognizability of palindromes on Turing machines. *Journal of Soviet Mathematics*, 15:68–77, 1981. (cit. on p. 86).

[271] Dina Sokol and Justin Tojeira. Speeding up the detection of tandem repeats over the edit distance. *Theoretical Computer Science*, 525:103–110, 2014. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2013.04.021. Advances in Stringology. (cit. on p. 128).

[272] Dina Sokol, Gary Benson, and Justin Tojeira. Tandem repeats over the edit distance. *Bioinformatics*, 23(2):e30–e35, 01 2007. doi: 10.1093/bioinformatics/btl309. (cit. on p. 128).

[273] Tatiana Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *Proc. of CPM*, pages 223–234, 2013. doi: 10.1007/978-3-642-38905-4_22. (cit. on p. 25).

[274] Teresa Anna Steiner. Differentially private approximate pattern matching. In *Proc. of ITCS*, pages 94:1–94:18, 2024. doi: 10.4230/LIPICS.ITCS.2024.94. (cit. on p. 40).

[275] Robert Susik, Szymon Grabowski, and Sebastian Deorowicz. Fast and simple circular pattern matching. In *Man-Machine Interactions 3*, pages 537–544, 2014. ISBN 978-3-319-02309-0. (cit. on pp. 17 and 26).

[276] Yuka Tanimura, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, Simon J. Puglisi, and Masayuki Takeda. Deterministic Sub-Linear Space LCE Data Structures With Efficient Construction. In *Proc. of CPM*, volume 54 of *LIPIcs*, pages 1:1–1:10, 2016. ISBN 978-3-95977-012-5. doi: 10.4230/LIPIcs.CPM.2016.1. (cit. on p. 63).

[277] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space LCE data structure with constant-time queries. In *Proc. of MFCS*, 2017. doi: 10.4230/LIPICS.MFCS.2017.10. (cit. on p. 63).

[278] I Tomohiro. Longest common extensions with recompression. In *Proc. of CPM*, volume 78, page 18, 2017. doi: 10.4230/LIPICS.CPM.2017.18. (cit. on p. 63).

[279] Esko Ukkonen. On approximate string matching. In *Proc. of FCT 1983*, volume 158 of *LNCS*, pages 487–495, 1983. doi: 10.1007/3-540-12689-9_129. (cit. on pp. 155, 158, 159, and 160).

[280] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. (cit. on p. 12).

[281] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10:308–315, 1975. (cit. on pp. xv and 4).

[282] Paul Valiant. Testing symmetric properties of distributions. In *Proc. of STOC*, pages 383–392, 2008. (cit. on p. 81).

[283] Dirk Van Gucht, Ryan Williams, David P. Woodruff, and Qin Zhang. The communication complexity of distributed set-joins with applications to matrix multiplication. In *Proc. of PODS 2015*, page 199–212, 2015. ISBN 9781450327572. doi: 10.1145/2745754.2745779. (cit. on p. 65).

[284] Peter Weiner. Linear pattern matching algorithms. In *Proc. of SWAT*, pages 1–11, 1973. doi: 10.1109/SWAT.1973.13. (cit. on pp. 11 and 12).

[285] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi: 10.1016/0020-0190(83)90075-3. (cit. on p. 26).

[286] Andrew Chi-Chin Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proc. of FOCS*, pages 222–227. IEEE, 1977. doi: 10.1109/SFCS.1977.24. (cit. on pp. 91 and 102).

[287] Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and control*, 10(2):189–208, 1967. (cit. on pp. xv and 4).