École Normale Supérieure de Lyon

Leçons d'Informatique
Agrégation 2022

Marin Malory, Sorci Émile, Rousseau Guillaume, Bertrand Jules

Table des matières

I Leçons **9**

1 Exemples de méthodes et outils pour la correction des programmes. **11**

2 Paradigmes de programmation : impératif, fonctionnel, objet. Exemples et applications. **17**

3 Tests de programme et inspection de code. **23**

4 Exemples de structures de données. Applications. **29**

5 Implémentations et applications des piles et des files. **37**

6 Implémentations et applications des ensembles et des dictionnaires. **43**

7 Accessibilité et chemins dans un graphe. Applications. **49**

8 Algorithmes de tri. Exemples, complexité et applications. **55**

9 Algorithmique du texte. Exemples et applications. **61**

10 Arbres : représentations et applications. **67**

11 Exemples d'algorithmes d'approximation et d'algorithmes probabilistes. **75**

12 Stratégies algorithmiques (dont glouton, diviser pour régner, programmation dynamique, retour sur trace). **81**

13 Algorithmes d'ordonnancement de tâches et de gestion de ressources. **87**

14 Gestion et coordination de multiples fils d'exécution. **93**

15 Hiérarchie mémoire. Structure et performances. **97**

16 Mémoire : du bit à l'abstraction vue par les processus. **103**

17 Problèmes et stratégies de cohérence et de synchronisation. **109**

18 Stockage et manipulation de données, des fichiers aux bases de données. **115**

19 Fonctions et circuits booléens en architecture des ordinateurs. **121**

20 Principes de fonctionnement des ordinateurs : architecture, notions d'assembleur. **127**

21 Échanges de données et routage. Exemples. **133**

22 Modèle relationnel et conception de bases de données. **139**

23 Requêtes en langage SQL. **147**

24 Exemples d'algorithmes d'apprentissage supervisés et non supervisés. 153

25 Analyses lexicale et syntaxique. Applications. 159

26 Classes P et NP. Problèmes NP-complets. Exemples. 167

27 Décidabilité et indécidabilité. Exemples. 173

28 Formules du calcul propositionnel : représentation, formes normales, satisfiabilité. Applications. 179

29 Langages rationnels et automates finis. Exemples et applications. 187

II Développements 193

1 Correction du balayage de Graham 195

2 Optimalité du glouton sur les matroïdes 199

3 B-arbres 201

4 Construction d'un tas en temps linéaire et tri par tas 205

5 Complexité moyenne de la recherche dans une table de hachage 209

6 Analyse du tri rapide randomisé 211

7 Distance d'édition 213

8 Automate des motifs 217

9 Algorithme d'approximation pour un problème de routage 221

10 Introduction à la méthode probabiliste 225

11 Algorithme CYK 227

12 Calcul de $\mathrm{premier}$ en analyse lexicale 229

13 Voyageur de commerce 233

14 2-SAT est linéaire 237

15 Théorème de compacité et application 241

16 Théorème de Myhill-Nérode 243

17 Performance de l'algorithme des k-moyennes 245

18 Un algorithme d'approximation pour un problème de clustering 247

19 Résolution d'un exercice avancé de SQL 249

20 Un algorithme d'approximation glouton pour un problème d'ordonnancement 253

21 Algorithmes onlines de remplacement de pages 257

22 Construction d'un additionneur à retenue anticipée 261

23 Recherche de chemin critique dans un circuit booléen 263

24 Validation croisée 265

25 Vérification du produit de matrice 267

26 Algorithme de Peterson 269

27 Algorithme d'ordonnancement online 273

28 Décidabilité et langages rationnels 275

29 Théorème de Rice 279

30 Codage de Huffman 281

Préface

Présentation

Enfin une agrégation d'informatique ! Il aura fallu attendre 2022 pour la création d'une agrégation entièrement dédiée à l'informatique, remplaçant ainsi l'ancienne agrégation de mathématiques option informatique. Avec elle suit une épreuve orale emblématique : la leçon.

De manière succincte, une leçon consiste à proposer un plan détaillé au jury après 4h de préparation sur un sujet tiré aléatoirement. Les dix premières minutes de l'oral consiste en une défense du plan dans lequel sera proposé deux développements. (Les dix premières minutes donnent lieu à la défense du plan ....) Le jury en choisira alors un que vous présenterez pendant une vingtaine de minutes. Pour plus de détails sur cette épreuve ainsi que sur l'agrégation d'informatique en général, je vous ramène au site de l'agrégation `https://agreg-info.org/`.

Nous vous proposons ici un ensemble de plans et développements préparés lors de la première session de 2022 par des membres de la promotion de l'ENS de Lyon.

Public visé

De manière évidente, ce document est avant tout destiné aux personnes préparant ou allant préparer l'agrégation d'informatique. Il est nécessaire pour cela d'avoir déjà un certain recul sur l'ensemble des notions présentées, le réel intérêt du document étant dans les choix et l'organisation des thèmes abordés, ainsi que dans les références utilisées.

Attention, une leçon doit rester personnelle, adaptée à vous et vos qualités. Il est inutile de reprendre telle qu'elle une leçon présentée ici puisque vous serez incapable de défendre de manière efficace un plan qui n'est pas le votre. Ces leçons doivent juste vous servir à ne pas partir de zero dans votre travail, comme nous avons eu à le faire pour cette première session.

Contenu

Ce document propose pour chaque leçon un plan détaillé ainsi que deux développements possibles. Il est important de noter que ces leçons et développements ont été écrits lors de la première session, sans recul sur les années précédentes et sans rapport de jury. Il sera donc important pour le lecteur de prendre en compte ce rapport pour modifier voir complètement changer ces plans. Enfin, l'ENS de Lyon recrutant avant tout des personnes venant d'un parcours très théorique, beaucoup de leçons et développement sont influencés par nos compétences (ou non-compétences). Nous demandons ainsi au lecteur à la fois de la méfiance et de la compréhension vis-à-vis des plans proposés dans ces leçons (particulièrement pour les leçons 18 et 21). À l'inverse, pour les personnes venant d'un parcours moins théorique, il est possible que nous allions trop loin dans certains concepts.

Remerciements

L'ensemble de ces leçons et développements ont été travaillé à l'École Normale Supérieure de Lyon, dans le cadre d'une formation dédiée à la préparation de l'agrégation d'informatique. Nous tenons à remercier l'ensemble des étudiants ayant préparé les oraux : Jules Bertrand, Luc Lapointe, Pierre-Marie Marcille,

Malory Marin, Guillaume Rousseau et Émile Sorci. Nous remercions aussi toutes les personnes nous ayant accompagné au cours de cette année, soit en examinant nos leçons ou soit en en tant que professeurs : Pascal Koiran, Simon Iosti, Florent De Dinechin, Valentin Bartier, Etienne Mauffret, Yves Robert, Rémi Pellerin, Eddy Carron, Émile Hazard, Natacha Portier, Nicolas Chappe, Angela Bonifati, Frederic Prost, Aurelien Garivier, Laurent Lefèvre et Christophe Alias.

Enfin, nous remercions Sasha Darmon et Pierre Marrec pour la relecture finale.

Malory Marin

Première partie

Leçons

Leçon 1

Exemples de méthodes et outils pour la correction des programmes.

Auteur·e·s: Bertrand Jules, Marin Malory
Niveau : L1-L2
Pré-requis : Algorithmique de base
Références : [Albert, 1998],[Nielson and Nielson, 2007],[Cormen et al., 2009]

I Introduction

Lorsqu'on programme, il est essentiel de s'assurer de la correction des algorithmes implantés. Tester notre programme permet seulement de prouver que notre algorithme n'est pas correct, alors on développe ici des méthodes et outils permettant de démontrer la correction de programmes.

Définition 1.1 (Terminaison) *Un programme P* **termine** *sur l'entrée x si le calcul de $P(x)$ nécessite un nombre fini d'étapes.*

Définition 1.2 (Spécification) *La* **spécification** *d'un programme P est la donnée de son domaine (les arguments de la fonction) ainsi que l'ensemble des résultats attendus.*

Exemple 1.1 *Le programme P prend en entrée un entier x et retourne l'entier $2x$ est une spécification de P. Plus formellement, on pourra écrire $\forall x \in \mathbb{N}, P(x) = 2x$.*

Définition 1.3 (Correction) *Un programme P est* **partiellement correcte** *si, sur l'entrée x, P retourne une valeur conforme à sa spécification lorsqu'il termine.*
De plus, si P termine sur toute entrée, on dit qu'il est **totalement correcte**.

terminaison + correction partielle = correction totale

II Terminaison

On cherche ici à montrer la terminaison d'un programme. On distinguera le cas des algorithmes itératifs et récursifs.

A Le cas des algorithmes itératifs

Dans un algorithme itératif, seul les boucles peuvent causer une non-terminaison. Ainsi, la plupart du temps, on devra s'intéresser à chaque boucle de noter programme. Pour montrer la terminaison, on utilisera la notion d'**ordre bien fondé**.

Définition 1.4 *Un ensemble $(E, \leq)$ est dit bien fondé s'il n'existe pas de suite infinie d'éléments de E strictement décroissante.*

Exemple 1.2 *$\mathbb{N}$ muni de la relation d'ordre naturelle est bien fondé, et il en est de même pour $\mathbb{N}^2$ muni de l'ordre lexicographique.*

Définition 1.5 *Un variant de boucle est une suite $(v_n)_{n\in\mathbb{N}}$ strictement décroissante à valeurs dans un ensemble bien fondé $(E, \leq)$ qui dépend de l'ensemble des valeurs des variables et du nombre d'itérations n passées.*

Pour montrer la correction d'un programme, il faut alors montrer que chaque boucle termine c'est-à-dire que cette boucle admet un variant.

Théorème 1.1 *Si chaque boucle d'un programme P admet un variant de boucle, le programme P termine.*

Remarque 1.1 *Toute boucle* `Pour` *admet un variant trivial. Considérons une boucle* `Pour i=a...b faire ...`*, alors la suite (v_n) définie par $v_n = b - i_n$ est un variant de boucle, où i_n est la valeur de i à l'itération n.*

Exemple 1.3 *Dans l'algorithme d'Euclide suivant, la suite définie par (b_n) où b_n est la valeur de b à l'itération n est un variant de boucle.*

Algorithme 1.1 : Euclide(x,y)

Données : $(x, y) \in \mathbb{N}^2$
Résultat : $PGCD(x, y)$
$a \leftarrow x$;
$b \leftarrow y$;
tant que $b \neq 0$ **faire**
  $a, b \leftarrow b, a \mod b$;
retourner a

B Le cas des algorithmes récursifs

On s'intéresse au cas des programmes récursifs. Ici, on dira que $f : A \to S$ est une fonction récursive si, pour $x \in A$, $f(x)$ dépend de certaines valeurs $f(y)$ pour $y \in A$.

Proposition 1.1 *Un ensemble $(E, \leq)$ est bien fondé si et seulement si toute partie non vide de E admet un élément minimal.*

Théorème 1.2 *Soit $f : A \to S$ une fonction récursive et $\varphi : A \to E$ avec $(E, \leq)$ bien fondé. Soit $B \subset A$ l'ensemble des minimiseurs de φ (cas de bases).*

Si, pour tout $b \in B$, $f(b)$ termine et si dans la définition de $f(x)$ (pour $x \in A$) n'apparaissent, en nombre fini, que des appels à $f(y)$ avec $\varphi(y) < \varphi(x)$, alors f termine pour tout $x \in A$.

Exercice 1.1 (Fonction d'Ackermann)

1. *Montrer que la fonction $A : \mathbb{N}^2 \to \mathbb{N}$ défini ci-dessous termine.*

$$A(n,m) = \begin{cases} m+1 & \text{si } n = 0 \\ A(n-1,1) & \text{si } m = 0 \\ A(n-1, A(n,m-1)) & \text{sinon} \end{cases}$$

2. *Déterminer pour tout $n \in \mathbb{N}$ les valeurs de $A(2,n)$ et $A(3,n)$. En déduire que pour tout $n \in \mathbb{N}$, $A(4,n) = 2^{2^{\cdots}} - 3$ avec $n-3$ deux empilés.*

C Les limites de la terminaison

Dans certains cas, on ne sait pas si une fonction termine sur toute ses entrées.

Exemple 1.4 (Suite de Syracuse) *On a seulement conjecturé que la fonction suivante termine.*

Algorithme 1.2 : Syracuse(x)

```
Données : entier x
Résultat : booléen b
si x = 1 alors
    retourner VRAI
sinon
    si x pair alors
        retourner Syracuse(x/2)
    sinon
        retourner Syracuse(3x + 1)
```

De plus, il existe un fameux théorème, démontré par Alan Turing, qui affirme qu'il n'existe pas de programme qui reçoit en entrée un autre programme, et qui renvoie VRAI si le programme termine et FAUX sinon.

Exercice 1.2 *Étant donné un programme P qui reçoit en entrée le code d'un programme Q et décide si Q termine, trouver une contradiction.*

III Correction

On va maintenant donner des méthodes permettant de montrer la correction d'un programme. De même, on sépare le cas itératif et récursif.

A Le cas des algorithmes itératifs

Comme pour la première partie, seules les boucles nécessitent des techniques pour montrer la correction.

Dans le cas itératif, l'idée de la correction est similaire à la terminaison. Au lieu d'exhiber un variant de boucle, on va cette fois exhiber un invariant de boucle.

Définition 1.6 *Étant donné un programme P et une boucle dans P, un invariant de boucle est un prédicat dépendant des valeurs des variables et de l'itération vérifiant les propriétés suivantes :*

- **initialisation :** *le prédicat est vrai avant l'entrée dans la boucle ;*
- **conservation :** *s'il est vrai avant une itération de boucle, il est vrai avant la prochaine itération de boucle*

À la fin de la boucle, le prédicat permet de donner une propriété utile pour prouver la correction du programme.

Exemple 1.5 (Correction de l'Algorithme d'Euclide) *On revient à notre exemple. Dans l'algorithme d'Euclide, on peut prendre l'invariant suivant : « $\mathbf{pgcd}(a,b) = \mathbf{pgcd}(x,y)$ ».*

- *Avant d'entrer dans la boucle, a (resp. b) contient la valeur x (resp. y) et donc le prédicat est vérifié.*
- *Ensuite, pour montrer la conservation, il suffit d'utiliser la propriété du pgcd $\mathbf{pgcd}(a,b) = \mathbf{pgcd}(b, a \bmod b)$.*

Enfin, à la sortie de la boucle, la condition du `Tant que` *est violée, et donc $b = 0$ et donc $\mathbf{pgcd}(a,b) = \mathbf{pgcd}(a,0) = a$. Via l'invariant, on a $\mathbf{pgcd}(a,b) = \mathbf{pgcd}(x,y) = a$. L'algorithme est ainsi totalement correcte.*

Calcul de l'enveloppe convexe. Étant donné un ensemble Q de points sur le plan, un problème classique consiste à trouver l'enveloppe convexe $\mathbf{EC}(Q)$, c'est-à-dire le plus petit polygone convexe qui contient l'ensemble des points de Q. Un algorithme classique pour calculer cette enveloppe convexe est le **balayage de Graham**.

Développement 1. Présentation du balayage de Graham et correction.

B Le cas des algorithmes récursifs

On reprend le formalisme donné dans la partie précédente.

Proposition 1.2 *Soit $(E, \leq)$ un ensemble bien fondé et p un prédicat sur les éléments de E. Si la propriété suivante est vérifiée :*

$$\forall x \in E, ((\forall y < x, p(y)) \Rightarrow p(x))$$

alors $\forall x \in E, p(x)$.

Remarque 1.2 *Si x est un élément minimal, alors $\forall y < x, p(y)$ est toujours vrai et donc il suffit de montrer $p(x)$.*

Théorème 1.3 *On reprend toutes les données du théorème 1.2 ainsi qu'un prédicat p_f sur les valeurs prises par f.*

Si pour tout $b \in B$, $p_f(b)$ et si pour tout $x \in A$, en notant $Y \subset A$ l'ensemble fini des arguments des appels récursifs de $f(x)$, on a :

- $\forall y \in Y, \varphi(y) < \varphi(x)$,
- $(\forall y \in Y, p_f(y)) \Rightarrow p_f(x)$

alors $\forall x \in A, p_f(x)$.

Exercice 1.3 *Programmez une fonction OCaml réalisant un tri rapide et montrer qu'elle est totalement correcte.*

Exercice 1.4 *Programmer une fonction récursive OCaml calculant $\binom{n}{p}$ via le triangle de Pascal et montrer qu'elle est totalement correcte.*

C Un algorithme indissociable de sa preuve

Jusqu'à présent, on prouvait les programmes après les avoir écrits. Pour certains paradigmes de programmations, comme la **programmation dynamique** et **diviser pour régner**, c'est une relation de récurrence qui motive l'algorithme. Dans ce cas, la preuve de correction de notre algorithme est indissociable du programme lui-même.

Exemple 1.6 *Pour calculer la distance d'édition entre deux chaînes de caractères en temps raisonnable, on exhibe d'abord une relation de récurrence complexe avant d'écrire l'algorithme.*

D Optimalité des algorithmes gloutons

Les algorithmes gloutons permettent parfois d'obtenir une solution optimale à un problème d'optimisation, comme l'algorithme de Kruskal permettant d'obtenir un arbre couvrant de poids minimal.

Pour montrer la correction partiel d'un tel algorithme, il fait alors montrer que notre algorithme retourne bien la solution optimale. Cette optimalité provient parfois de la structure même des objets étudiés.

Développement 2. Dans le cadre des arbres couvrants, on peut montrer que l'ensemble des forêts d'un graphe forment un **matroïde**, structure sur lequel le glouton est optimal.

IV Vers de la preuve automatique

A Exemple de la dichotomie

Dans le code classique de la dichotomie, pour calculer le milieu on utilise la formule $m = (l + h)/2$. Or, un problème peut occurrer lorsque $l + h$ cause un débordement arithmétique (c'était notamment le cas dans la fonction `binarySearch` de Java).

> **Remarque 1.3** *Pour corriger ce problème, il suffit d'utiliser la formule $m = l + (h - l)/2$.*

Lors d'une démonstration de la correction manuelle, on peut négliger ce genre de problèmes venant de la représentation machine.

B Triplet de Hoare

Pour se tourner vers une automatisation des preuves, on va avoir besoin de logiques de programmes. Une logique de programmes fournit un **langage de spécification** et des **principes de raisonnement** sur les comportements du programme.

On peut alors formaliser la définition de spécification.

Définition 1.7 *Les* **spécifications** *se présentent généralement comme des* **assertions logiques** *portant su le programme :*

- **préconditions** *: hypothèses sur les entrées (paramètres des fonctions ; valeurs initiales des variables)*
- **postconditions** *: garanties sur les sorties (résultats de fonction ; valeurs finales des variables)*

— **invariants** *: garanties sur l'état en un point du code (invariants de boucles, de structures de données,...)*

Définition 1.8 (Triplet de Hoare)

Un **triplet de Hoare faible**, *noté* $\{P\}c\{Q\}$ *où* P *et* Q *sont respectivement la précondition et la postcondition et* c *est une commande, signifie :*

> *≪ Si la commande* c*, démarrée dans un état initial satisfaisant* P*, termine, alors l'état final satisfait* Q*. ≫*

Un **triplet de Hoare fort**, *noté* $[P]c[Q]$ *où* P *et* Q *sont respectivement la précondition et la postcondition et* c *est une commande, signifie :*

> *≪ La commande* c*, démarrée dans un état initial satisfaisant* P*, termine dans un état final satisfaisant* Q*. ≫*

Exemple 1.7 (Frama-C) *Des outils comme* **Frama-C** *permettent de spécifier un programme écrit en C et de démontrer automatiquement la validité des triplets de Hoare associés.*

Exemple 1.8 (COQ) *Pour OCaml, le logiciel de preuve formel COQ permet aussi de prouver la correction de fonctions de manière semi-automatique. Il donne un langage formel permetant d'écrire des définitions mathématiques, des algorithmes avec une syntaxe très similaire à OCaml et des théorèmes.*

C Syntaxe et sémantique des programmes

Pour terminer notre formalisation, il convient de distinguer la syntaxe et la sémantique des langages de programmation.

La **syntaxe** définit la structure grammaticale d'un programme. Ainsi, l'analyse syntaxique du programme ≪ `z :=x ; x:=y ; y:= z` ≫ nous dit que ce programme est composé de trois affectations séparées par des ≪ ; ≫.

La **sémantique** s'intéresse à la signification d'un programme grammaticalement correcte. Sur l'exemple précédent, une analyse sémantique nous permet de comprendre que ce programme échange les valeurs contenues dans `x` et `y`. On distinguera trois sémantiques :

— la **sémantique opérationnelle** : expliquer *comment* s'exécute un programme ;

— la **sémantique dénotationnelle** : explique la *réalisation* du programme ;

— la **sémantique axiomatique** : donne des propriétés spécifiques sur l'exécution du programme.

Ces trois sémantiques permettent de prouver formellement des résultats de correction et les triplets de Hoare vu précédemment sont très liés à la sémantique axiomatique.

Leçon 2

Paradigmes de programmation : impératif, fonctionnel, objet. Exemples et applications.

Auteur·e·s: Rousseau Guillaume, Marin Malory
Niveau : L1/L2
Pré-requis : Notions de base de programmation
Références : [Narbel, 2005], [Lesesvre et al., 2020]

Introduction

Revenons à une définition de base , qu'est-ce que la programmation informatique ? Cela consiste à concevoir une solution algorithmique à un problème et à la transcrire dans un langage informatique.

Le **paradigme de programmation** est alors une manière d'approcher un problème et formuler une solution dans un langage approprié.

Il existe de nombreux langages de programmations, et chaque langage a son propre modèle de conception avec sa propre syntaxe et ses propres caractéristiques (Python, C, OCaml, Rust, JavaScript,...). Cependant, ils peuvent présenter beaucoup de liens et de ressemblances, et peuvent plus ou moins adapté selon le paradigme choisi.

I Monoparadigme

A Impératif

Définition 2.1 *La* **programmation impérative** *consiste à résoudre un problème via une suite d'instructions. En particulier, on décrit la structure de contrôle du programme (variables, état de la mémoire, etc).*

Exemple 2.1 *Voici un simple programme impératif écrit en python.*

```
x=1
y=x+3
if y< 5 :
        print(y)
else :
        print(x)
```

Beaucoup de langages sont basés sur ce paradigme, le plus bas niveau étant le code machine (assembleur). Ensuite, Python, C et JavaScript sont aussi adaptés ce paradigme.

B Déclaratif

Définition 2.2 *La* **programmation déclarative** *consiste à résoudre un problème en décrivant la solution (sans dire clairement comment l'obtenir).*

Exemple 2.2

- *Une requête SQL décrit la solution sans donner l'algorithme de recherche dans la base de donnée.*
- *Pour écrire un fichier Latex, on décrit seulement le contenue du fichier et on donne sa structure. HTML et CSS fonctionnent aussi de cette manière.*

On remarque alors qu'un langage déclaratif nécessite beaucoup plus d'**abstractions** qu'un langage déclaratif. Une abstraction est un objet que l'on peut manipuler sans se soucier de l'implémentation (une fonction `numpy` et python par exemple). On peut alors voir un langage de programmation comme un ensemble d'abstractions.

Une branche importante du paradigme déclaratif constitue le paradigme fonctionnel, qui permet de définir des fonctions.

C Fonctionnel

Définition 2.3 *La* **programmation fonctionnelle** *consiste à composer le programme de fonctions (au sens mathématiques).*

Exemple 2.3 (Déclaration de fonction en Python et OCaml) *On souhaite définir une fonction permettant d'incrémenter une variable entière, et l'appliquer à 0.*

En Python,

```
def f(x) :
        return x+1
f(0)
```

En OCaml,

```
let f x = x +1 ;;
f 0 ;;
```

L'une des forces du paradigme fonctionnel est de considérer une fonction comme un autre type, permettant de les passer en argument d'une autre fonction par exemple. On appelle cela les **fonctions d'ordre supérieur**.

Exemple 2.4 *On souhaite écrire une fonction OCaml qui reçoit deux fonctions et retourne la composée de ces deux fonctions.*

```
let comp f g = (fun x -> f (g x) ) ;;
```

Les fonctions d'ordres supérieur peuvent aussi être utilisées par Python ou C par exemple.

Ce paradigme est particulièrement adapté lorsqu'on manipule des structures naturellement récursive (ou inductive) comme les arbres ou les entiers. Par exemple, le logiciel Coq de preuve assisté par ordinateur est codé en OCaml. Ce logiciel permet aussi de créer des compilateurs certifié de C, comme CompCert.

Le fonctionnel peut aussi être vu comme du déclaratif puisqu'on utilise les fonctions comme des boîtes à imbriquer et connecter pour décrire la solution.

Exemple 2.5 *La fonction factorielle peut se définir par induction sur les entiers par* $0! = 1$ *et* $n! = n \times (n-1)!$*. La fonction factorielle peut alors s'écrire directement en OCaml :*

```
let rec fact n = match n with
        | 0 -> 1
        |n -> n * (fact (n-1)) ;;
```

Dans certains cas, il s'agit même de la seule manière connue de programmer une fonction. Par exemple, la fonction d'Ackermann $A : \mathbb{N}^2 \to \mathbb{N}$ défini ci-dessous :

$$A(n,m) = \begin{cases} m+1 & \text{si } n = 0 \\ A(n-1,1) & \text{si } m = 0 \\ A(n-1, A(n,m-1)) & \text{sinon} \end{cases}$$

ne présente par de forme analytique.

Exercice 2.1 *Écrire une fonction OCaml qui implémente la fonction d'Ackermann, et montrer qu'elle termine.*

D Orienté Objet

Définition 2.4 *La* **programmation orienté objet** *consiste à résoudre le problème en construisant un certain nombre d'objets que l'on fait interagir. Un objet est programmé dans une* **classe** *qui peut contenir deux types d'informations :*

- *les attributs : informations qui caractérisent l'objet ;*
- *les méthodes : fonctions caractéristiques de l'objet.*

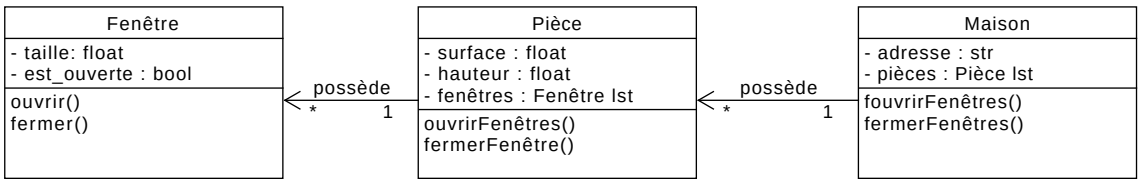
Exemple 2.6 *En python, le package numpy propose les objets* `numpy.array` *que l'on crée via la commande* `a = numpy.array([...])`*. Un tel objet possède des attributs comme sa taille (*`a.size` *mais aussi des méthodes (a.sort() par exemple).*

Remarque 2.1 *Les méthodes peuvent aussi caractériser le comportement d'un objet avec un autre objet.*

En pratique, le programmeur interagit avec un objet seulement par ses méthodes. On parle d'**encapsulation**.

Un outil puissant pour représenter de manière schématique les classes d'un programme sont les diagrammes UML. Chaque classe est représentée par une boîte contenant ses attributs et ses méthodes. Les différentes interactions entre classes sont précisés par des flèches différentes.

Exemple 2.7 *On présente ici un schéma UML d'un programme contenant trois objets : maison, fenêtre et pièce.*



Lorsqu'un objet est un sous-ensemble d'un autre objet, on parle d'**héritage**. Ainsi, le premier objet hérite des attributs et méthodes de la seconde, auquel on peut ajouter des caractéristiques spécifiques.

II Paradigmes avancés

A Multiparadigme

En pratique, la plupart des langages ne se réduisent pas à un unique paradigme, mais en favorise certains par rapport à d'autres. Par exemple, il est possible de faire du fonctionnel en C, même s'il n'est pas du tout adapté pour cela.

Pourquoi utilisent-on plusieurs paradigmes ?

Abstraction. Le déclaratif n'est qu'une abstraction permettant de raisonner plus facilement. En cas d'opération complexe, le programmeur peut atteindre les limites des abstractions disponibles. De plus, dans la programmation orienté objet, les méthodes d'une classe sont aussi une abstraction qui peuvent être programmer en utilisant un autre paradigme. C'est pour cela que les langages permettant ce paradigme proposent aussi d'autres paradigmes.

Exercice 2.2 (TP) *Implémentation d'un jeu de rôle contenant plusieurs objets (joueur/personnage/monstres/armes).*

B Programmation concurrente

Définition 2.5 *La* **programmation concurrente** *consiste à diviser son programme en plusieurs fils d'exécution s'exécutant en parallèle.*

Ce paradigme peut être très utile lorsque son programme peut facilement être divisé en tâche parallèle.

Processus et Threads.

Définition 2.6 *Un* **processus** *est une abstraction par le système d'un programme en cours d'exécution.*
Un **thread** *ou* **fil d'exécution** *d'un processus est une exécution (état des registres et de la pile) pour un processus.*

Exemple 2.8 *Il y a au moins un processus par fenêtre ouverte.*

Ici, le multiparadigme est central puisque chacun des différents threads effectue une tâche bien définie, le plus souvent programmé avec un paradigme impératif.

Remarque 2.2

1. *Il existe de nombreux processus dont on n'a pas conscience, qui tourne continuellement en fond. Pour l'instant, on pourra considérer que tous les processus s'exécute en parallèle.*
2. *L'espace d'adressage entre threads d'un même processus est le même, ils partagent les variables globales. Les piles et états des registres sont cependant distincts.*

Manipulation. La librairie `pthread` en C permet de manipuler les threads.

Exercice 2.3 (TP)

1. *Manipulation des threads en C.*
2. *Écrire un programme où plusieurs threads se comptent en incrémentant un compteur partagé.*

Le deuxième exercice montre un non déterminisme : il y a besoin d'exclusion mutuelle.

Cohérence et synchronisation de processus

Définition 2.7 *Une* **condition de concurrence** *est une situation où deux threads ou processus exécutent une zone du code d'écriture ou de lecture dans sur une mémoire partagée et dont le résultat dépend de l'ordre d'exécution des instructions. Une telle zone de code est appelée* **section critique**.

Si plusieurs threads n'entrent jamais dans leur section critique en même temps, on dit qu'il y a **exclusion mutuelle**.

Exemple 2.9 *Dans le programme précédent où plusieurs threads se comptent, la section critique correspond à la ligne d'incrémentation du compteur. On peut alors identifier la condition de concurrence.*

Implémentation de l'exclusion mutuelle par attente active

Premier essai Une première solution consiste à utiliser une variable globale verrou que l'on incrémente pour entrer en section critique. On peut ici noter un problème d'atomicité de l'opération qui empêche d'avoir vraiment exclusion mutuelle.

Solution de Peterson L'algorithme de Peterson permet de résoudre le problème de l'exclusion mutuelle pour 2 threads.

Solution de Lamport L'algorithme de la boulangerie de Lamport permet de résoudre le problème de l'exclusion mutuelle pour p threads.

Développement 26. On montre certaines propriété qu'assure l'algorithme de Peterson.

Ces deux solutions présentent un problème majeur : il s'agit d'attente active. Le processeur exécute du code qui ne fait rien, et donc de la ressource est perdue.

Implémentation de l'exclusion mutuelle avec des mutex et sémaphores

Définition 2.8 *Un* **sémaphore** *est une variable qui stocke un entier qui représente le nombre de ressources d'un type disponible. Deux opérations sont possibles sur un tel entier : le prendre (décrémenter) s'il est non nul (sinon, on bloque tant qu'il est nul) ou le libérer de manière réciproque. Ces opérations sont* **atomiques**.

Un **mutex** *est un sémaphore binaire.*

Pour implémenter les sémaphores et mutex dans le système, on utilise des interruptions. Lorsqu'un thread bloque, il est placé dans un état bloqué special. L'ordonnanceur (cf partie suivante) n'essaye plus de l'exécuter tant qu'il n'a pas récupérer l'information disant que le mutex ou sémaphore a été libéré. La librairie `pthread` permet d'utiliser des mutex, et la librairie `semaphore` les sémaphores.

C Programmation orienté automate

On présente ici un dernier paradigme, qui consiste à utiliser les automates finis déterministe.

Définition 2.9 *La* **programmation orienté automate** *consiste à résoudre un problème en proposant un automate finis déterministe qui résout le problème, puis en l'implémentant.*

Exercice 2.4 *Proposer une implémentation en python d'un automate finis déterministe.*

Un résultat fameux, admis ici, est la décidabilité de l'**arithmétique de Presburger**. En effet, il existe un automate finis déterministe (avec beaucoup d'états) qui étant donné une formule logique du premier ordre sur les entiers munis de l'addition, décide si elle est vraie ou non.

Recherche de motif. Ce paradigme est aussi très utile en algorithmique du texte. Étant donné un texte T de longueur n et un motif M de longueur m, l'algorithme naïf décidant si M est une sous-chaîne de T est un $\mathcal{O}(nm)$. On peut mieux faire avec ce paradigme.
Développement 8. Recherche de motif à l'aide de l'automate des occurrences.

Leçon 3

Tests de programme et inspection de code.

Auteur·e·s: Marin Malory
Niveau : L2
Pré-requis : Algorithmique, Graphes
Références : [Myers et al., 2012]

I Introduction

Test de programmes. Dans le sens commun, tester notre programme a tendance à avoir l'objectif de montrer qu'il fonctionne bien. Cette définition n'est pas correcte.

Définition 3.1 *Tester est le processus qui consiste à exécuter un programme avec l'intention de trouver des erreurs.*

Cette définition a deux implications :

1. tester est un processus *destructif*, ce qui le rend compliqué (d'un point de vue psychologique) ;
2. un test est réussi lorsqu'il trouve une erreur, et non l'inverse.

La première question est alors : est-il possible de tester notre programme afin de trouver toutes les erreurs ? On étudie deux approches pour répondre à cette question.

Tests en boîte noire. La première méthode de test consiste à considérer notre programme comme une boîte noire.

Si quelqu'un veut trouver toutes les erreurs sur le programme, il faut alors tester toutes les entrées possibles. C'est déjà impossible lorsqu'il y a une infinité de donnée possible (ou même un très grand nombre).

Ce test doit donc s'effectuer sur un sous-ensemble des entrées possibles, et il n'y a aucun raison qu'il n'y ait pas une erreur spécifique pour un de ces cas.

Tests en boîte blanche. La seconde méthode de test nous permet d'avoir accès au code du programme. De ce code on peut extraire un **graphe de flot de contrôle**. Au lieu de tester l'ensemble des données d'entrées, on peut alors tester l'ensemble des chemins du graphe. Ceci est impossible dès lors que le graphe contient une boucle.

Remarque 3.1 *Tester l'ensemble des chemins n'est pas optimal. Lorsque par exemple le code contient une instruction de la forme* `if (a-b) < ϵ then ...` *au lieu de* `if |a-b| < ϵ then ...`*, tester le chemin une unique fois ne suffit pas.*

Ces deux méthode ne permettent pas de détecter toutes les erreurs, on va donc essayer de les combiner afin d'avoir des jeux On parle de tests raisonnables.

Principes de tests. On présente ici quelques principes nécessaires lorsqu'on créer un jeu de tests.

1. Une partie nécessaire d'un jeu de test est la définition claire des résultats attendus.
2. Un programmeur doit éviter de tester son propre programme.
3. Un jeu de test doit être écrit avec des conditions d'entrées invalides ou imprévues, mais aussi des conditions valides et prévus.
4. Vérifier si le programme fait ce qu'il doit faire et s'il ne fait pas ce qu'il est censé ne pas faire.
5. La probabilité qu'une erreur soit présente dans une portion du programme augmente avec le nombre d'erreurs déjà trouvées à cet endroit.

II Inspection de code

Avant de parler de méthodes de tests par ordinateurs, on va se concentrer sur des méthodes ≪ humaines ≫. Les bonnes pratiques de programmations permettent souvent de rendre le code lisible et de trouver des erreurs de manière plus efficace car trouvées plus tôt.

Inspection. Le but est ici de lire un code et de trouver d'éventuelles erreurs et non pas de les corriger. Une inspection de code se réalise le plus souvent en groupe, dans lequel se trouve le programmeur.

Définition 3.2 *Une inspection de code est un ensemble de procédures et de techniques de détection d'erreurs pour un groupe de lecture de code.*

Remarque 3.2 *Lorsque c'est le programmeur seul qui fait l'inspection, on parle plutôt de contrôle de code (desk-checking), la distinction étant dû à la différence d'efficacité.*

Cette méthode présente des avantages : lorsqu'une erreur est trouvé, c'est se nature même qui est trouvé, et non pas juste un symptôme ; et on repère souvent un ensemble d'erreurs. Une inspection de code se réalise en trois étapes :

1. les participants prennent connaissance du code à l'avance ;
2. le programmeur explique lui-même son code aux participants (et peut alors lui-même trouver des erreurs) ;
3. le programme est analysé via une liste d'erreurs classiques.

Une liste d'erreurs classiques peut être construite suivant ces points :

— Erreurs de référence de données (variables non initialisée, oublie d'une allocation de mémoire).
— Erreurs de déclaration de données (variable non déclarée, erreur de typage, homonymes).
— Erreur de calcul (division par zéro, somme d'entiers et de flottants).
— Erreurs de comparaisons (mauvais comparateur, mauvais opérateur booléen).
— Erreurs de contrôle de flot (terminaison des boucles)

Exécution pas à pas. Une autre méthode de test ≪ humain ≫ est de simuler l'exécution du programme. Elle permet notamment de trouver plus facilement les erreurs de logique du programmes (comme un branchement inutile).

Remarque 3.3 *Ces exercices sont aussi très bénéfiques au programmeur lui-même, qui reçoit un avis extérieur sur son code.*

III Jeu de tests

Comme on l'a vu précédemment, faire un jeu de test capable de trouver toutes les erreurs est impossible. **Quel sous-ensemble des tests possibles a la plus grande probabilité de détecter des erreurs ?** Pour construire un tel jeu de test, on va utiliser les deux approches vus précédemment.

Boîte noire	**Boîte blanche**
Partitions d'équivalences	Couverture des instructions
Analyse à valeurs bornées	Couverture de décisions
Graphe cause-effet	Couverture des conditions
Prédiction d'erreurs	Couverture des conditions-décisions
	Couverture de conditions multiples

L'approche classique consiste à d'abord faire un jeu de test en utilisant une approche boîte noire, puis de le compléter avec des méthodes boîte blanche.

A Tests en boîte blanche

Les tests en boîte blanche ont pour objectif de couvrir au maximum la logique du programme. Cela se traduit par un parcours de tous les chemins du programme, ce qui le rend impossible dès qu'il y a une boucle.

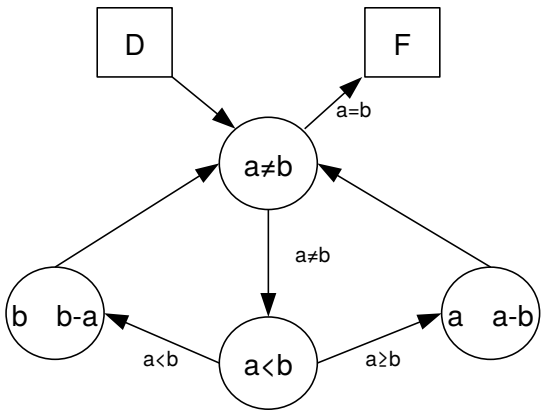
Définition 3.3 (Graphe de contrôle) *Étant donné un code de programme, le* **graphe de contrôle** *de ce code est un graphe orienté $G = (V, E)$ où :*

- *V est l'ensemble des instructions élémentaires du code (affectation, test conditionnel, etc), avec un sommet initial D et final F ;*
- *$ij \in E$ si et seulement si le programme peut passer directement de l'instruction i à l'instruction j. L'arc ij est étiquetée par la condition de saut (souvent un booléen).*

Exemple : PGCD. On prend ici l'exemple du calcul du pgcd.

Algorithme 3.1 : PGCD(a,b)

Données : $a, b \in \mathbb{N}^*$
Sorties : pgcd(a, b)
tant que $a \neq b$ **faire**
  si $a < b$ **alors**
    $b \leftarrow b - a$;
  sinon
    $a \leftarrow a - b$;
retourner a ;



Un test va parcourir le graphe de D à F. On peut alors donner un critère pour que l'ensemble du jeu de tests parcourt :

1. tous les sommets ;
2. tous les arcs ;

3. toutes les conditions ;
4. toutes les conditions-décisions ;
5. toutes les conditions multiples.

Exemple 3.1 (PGCD) *Un jeu de test pour le critère « tous les arcs » est* $\{(a = 1, b = 2), (a = 2, b = 1)\}$. *On remarque aussi qu'il respecte le critère tous les sommets.*

Exercice 3.1 (Hiérarchie) *Montrer que pour tout* $1 < i \leq 5$, *si un jeu de test respecte le critère* i, *alors il respecte le critère* $i + 1$.

B Tests en boîte noire

Nous verrons principalement deux méthodes : par partitions d'équivalence et l'analyse à valeurs bornées. La première consiste à partitionner les données en entrée en classes d'équivalence, puis le jeu de test doit au moins tester une fois chaque classe. La seconde, fortement liée à la première, consiste à tester toutes les données au bord des classes d'équivalence.

Exercice 3.2 *Proposer un jeu de test en boîte noire pour un programme calculant le PGCD de deux entiers positifs.*

IV Test de modèle

Dans certains cas, comme en apprentissage machine, les erreurs sont intrinsèques au problème. La principale différence entre le test d'un algorithme classique vu jusqu'alors et un algorithme d'apprentissage vient de l'espace des données :

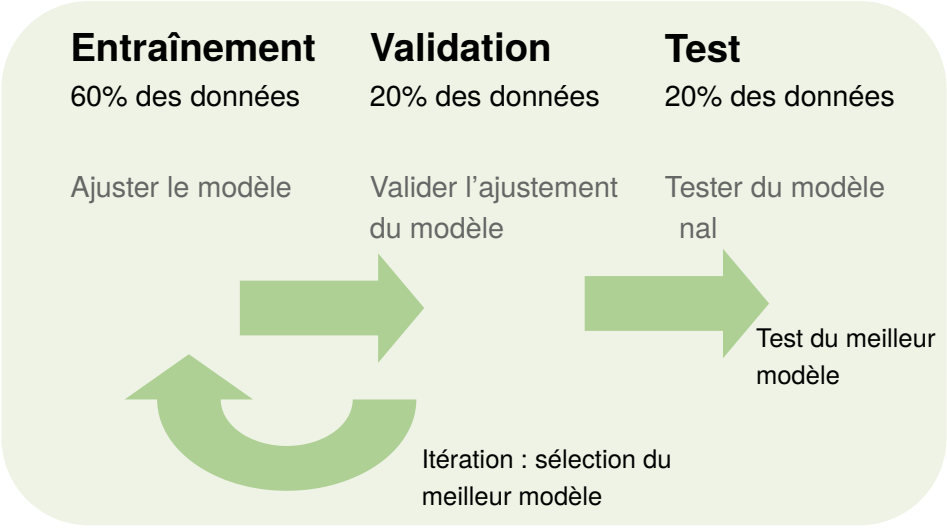
— pour un algorithme classique, on connaît pour chaque entrée possible, le comportement voulu ;
— tandis qu'en apprentissage machine, on connaît le résultat voulu pour un sous-ensemble des entrées possible, appelé jeu d'**entraînement**.

Il y a alors deux niveaux de lectures de notre algorithme d'apprentissage :

— la vue d'un algorithme classique, qui peut être prouvé et testé,
— la vue d'un modèle (de prédiction, de classification) qui peut être testé afin de vérifier s'il est pertinent.

Modèle. On suppose qu'on dispose d'un algorithme d'apprentissage fonctionnant sur un ensemble d'entraînement S et un ensemble de paramètre Θ. Le but est de trouver les meilleurs paramètres possibles.

Diviser ses données. Puisque dans notre cas, on n'a un choix limité de donnée pour tester notre algorithme une division s'impose. En effet, il nous faut des données pour **entraîner**, pour **optimiser** les paramètres (**validation**) et enfin pour **tester** notre algorithme final. La méthode la plus simple consiste à couper notre jeu de donnée en 3 (60%, 20%, 20%).



Développement 24. Il existe une autre méthode pour optimiser nos paramètres : la validation croisée.

Mesures de performances. Lorsqu'on teste notre algorithme (en phase de vérification et de test), plusieurs choix de métriques s'offrent à nous :

— en régression : moyenne des erreurs absolues, moyenne des erreurs quadratiques, etc

— classification : nombre d'erreurs, précision, sensibilité, etc.

V Tests efficaces

Les tests d'un programme peuvent s'avérer coûteux en terme de complexité algorithmiques. Par exemple, prenons un algorithme $\mathcal{A}$ de produit de matrice . À chaque exécution de notre algorithmes sur les entrées A et B, il faut vérifier que $\mathcal{A}(A, B) = AB$. Il faut donc réaliser un produit de matrice avec un algorithme dont on est déjà sûr. Ainsi, à chaque test, on réalise un $\mathcal{O}(n^3)$ opérations élémentaires en plus de l'exécution de l'algorithme. Il existe des manières plus efficaces pour réaliser ce test.

Développement 25. Vérification du produit de matrice efficace.

Leçon 4

Exemples de structures de données. Applications.

Auteur·e·s: Sorci Émile
Niveau : L3
Pré-requis : Structures de données de base (liste, tableau, pile et file), algorithmique de base, notion de graphe et d'arbre.
Références : [Cormen et al., 2009],[Gaudel et al., 1990],[Dasgupta et al., 2008]

Note pour la défense du plan : L'idée de la leçon est de partir des algorithmes. On explique les algorithmes et on remarque quels sont nos besoins au niveau des structures de données (quelles sont les opérations fréquentes, qu'est ce qui pourrait être coûteux) et à partir de cela on propose de nouvelles structures.

I Introduction

A Motivation

Dans cette leçon nous apprenons à concevoir, utiliser et analyser des structures de données avancées pour développer des algorithmes efficaces.

B Structure de donnée abstraite et implémentations [Gaudel et al., 1990]

Définition 4.1 *Une* **structure de donnée abstraite** *est une liste d'opérations que l'on peut effectuer sur un ensemble de données.*

Exemple 4.1 *La structure de donnée de pile FIFO (First-In-First-Out) est définie par les opérations* `empiler`, `dépiler` *et* `est_vide` *où :*

- `empiler` *prend en entrée une pile* `P` *et un élément* `x` *et empile l'élément* `x` *sur la pile* `P`.
- `depile` *prend en entrée une pile non vide* `P` *et enlève le dernier élément* `x` *empilé sur* `P` *et le renvoie.*
- `est_vide` *prend en entrée une pile* `P` *et renvoie un booléen :* `Vrai` *si la pile est vide et* `Faux` *sinon.*

Exercice 4.1 *Donner une spécification possible pour la structure de donnée abstraite de file FILO (First-In-Last-Out) et dictionnaire.*

Remarque 4.1 *La spécification que la pile ne doit pas être vide pour pouvoir dépiler est appelée une* **précondition**. *Le comportement qu'adopte l'opération dépile dans le cas de la pile vide n'est pas spécifié.*

Définition 4.2 *Une* **implémentation d'une structure de donnée abstraite** *est la spécification technique qui définit comment les données sont stockées dans la machine et qui fournit l'algorithme pour chacune des opérations définies par la structure abstraite.*
On parlera simplement de **structure de données**. *On peut alors définir la complexité en temps et en espace des opérations dans le pire cas et en moyenne ainsi que l'espace occupé par une instance de la structure.*

Exemple 4.2 *La structure de donnée abstraite de pile peut être implémentée par une liste chaînée. On donne la complexité des opérations ainsi que l'espace de stockage utilisé par la structure.*

Exercice 4.2 *Donner une implémentation de la structure abstraite de file qui utilise deux listes chaînées ainsi que la complexité des opérations dans le pire cas et l'espace utilisé.*

C Un premier exemple : les graphes

Définition 4.3 (Structure de donnée abstraite des graphes)

- `Est_arête(G, u, v)`
- `Voisins(G, u)`
- `Ajouter_sommet(G, x)`
- `Enlever_sommet(G, x)`
- `Ajouter_arête(G, u, v)`
- `Enlever_arête(G, u, v)`

Implémentation 4.1 *Implémentation avec des listes d'adjacence.*
Avantages : `Voisins(G, u)` *en* $O(1)$ *et espace* $O(|A| + |S|)$
Inconvénients : `Est_arête(G, u, v)` en $O(deg_G(u))$

Implémentation 4.2 *Implémentation avec des matrices d'adjacence.*
Avantages : `Est_arête(G, u, v)` en $O(1)$
Inconvénients : `Voisins(G, u)` en $O(|S|)$ et espace en $O(|S|^2)$

On préférera une implémentation avec des listes d'adjacence si on fait de nombreux appels à `Voisins(G, u)`.

Cas de l'algorithme de Floyd Warshall : On préférera une implémentation avec des matrices d'adjacence, structure que l'algorithme utilise naturellement.

Exercice 4.3 *Pour tout graphe orienté à n sommets, on définit un puits comme étant un sommet de degré entrant égal à* $n - 1$ *et de degré sortant nul. En admettant que le graphe est représenté par sa matrice d'adjacence, montrer que l'on peut déterminer si le graphe admet un puits en* $\mathcal{O}(n)$.

Remarque 4.2 *Ce résultat est surprenant car, dès que l'on représente les graphes par matrice d'adjacence, les algorithmes ont tendance à avoir une complexité au moins* $\mathcal{O}(n^2)$.

D Les dictionnaires

Soit U un ensemble d'éléments appelés valeurs, et K un ensemble d'éléments appelés clés. Un **dictionnaire** sert à stocker des couples (k, x) avec $k \in K, x \in U$.

Définition 4.4 (Structure de donnée abstraite pour un dictionnaire)

— `Insérer(D,x,k)`
— `Recherche(D,k)`
— `Supprimer(D,k)`

Il y a de nombreuses implémentations possibles pour les dictionnaires (tableaux, arbres de recherche, table de hachage) plus ou moins pertinente selon le contexte.

B-Arbres. Un B-arbre est une implémentation d'un dictionnaire sous la forme d'un arbre équilibré. Son utilisation est très pertinente dans le cadre d'une hiérarchie mémoire.
Développement 3. B-arbres.

II Les files de priorité

A Algorithme de Dijkstra [Dasgupta et al., 2008]

Algorithme 4.1 : Dijkstra(G,s)

Données : Un graphe pondéré G et un sommet s
Sorties : Plus courts chemins de s vers les autres sommets
pour $v \in V$ **faire**
  $d[v] \leftarrow +\infty$;
distance[s] $\leftarrow 0$;
$F \leftarrow$ `StructureVide()` ;
Insérer v dans F avec la clé 0 ;
tant que F *non vide* **faire**
  Extraire u de F de clé minimal d_u ;
  pour v *voisins de* u *dans* G **faire**
    $d \leftarrow d_u + w(u,v)$;
    si $d \leq$ *distance[v]* **alors**
      si *distance[v]* $= +\infty$ **alors**
        Insérer v dans F avec la clé d ;
      sinon
        $\#v$ est déjà dans F avec une clé $> d$;
        Diminuer la clé de v dans F à d ;
retourner *distance*

L'algorithme de Dijkstra sélectionne itérativement et de façon gloutonne les sommets les plus proches de s qui sont accessibles par le sommet courant. Il utilise un front de sommets éligibles au cours de son exécution.

Remarque 4.3 *On ne s'intéresse pas ici à la correction de l'algorithme mais aux structures de données à utiliser.*

Besoin de deux structures de données : une pour le graphe et une seconde que l'on appelle file de priorité.

Nous allons évidemment utiliser une implémentation du graphe avec des listes d'adjacence car on a besoin de parcourir les sommets adjacents.

B La structure de donnée

Définition 4.5 *Une* **file de priorité** *est une structure qui permet d'organiser un ensemble d'éléments munis de clés, l'ensemble de clés étant ordonné. Une file de priorité* `Q` *est définie pour tout élément* `x` *associé à une clef* `k` *par les opérations* `Inserer(Q, x, k)`, `Extraire_min(Q)`, `Est_vide(Q)` *et* `Réduire_clé(Q, x, k)`.

C Implémentations

Implémentation de file de priorité avec un tableau Dans un tableau on stocke dans la case i la clé de l'élément i.

Exercice 4.4 *Donner une implémentation des opérations qui définissent une file de priorité ainsi que leurs complexités dans le pire cas.*

On remarque que l'implémentation a des performances faibles notamment pour la fonction `Extraire_min(Q)`.

Implémentation de la structure de file de priorité avec un tas

Définition 4.6 *Un* **tas-min** *(resp. max) est un arbre binaire complet gauche dont les noeuds sont étiquetés par des éléments munis de clés qui vérifie la propriété du tas-min : la clé d'un nœud est inférieure aux clés de ses enfants.*

Exemple 4.3 *On donne un tas-max pour la liste [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]. On donne un tas-min avec ces éléments ainsi que plusieurs exemples d'arbres qui ne sont pas des tas.*

On utilise un tableau indexé de 1 à n pour stocker un tas de n éléments. On accède aux enfants d'un noeud d'indice i en prennant les éléments d'indice $2i$ et $2i+1$ et on accède au parent en prenant l'élément d'indice $\lfloor i/2 \rfloor$.

Exemple 4.4

1. $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ *vérifie la structure de tas-max ;*
2. *On montre qu'il s'agit du parcours en largeur de l'arbre.*
3. $[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$ *ne vérifie pas la structure de tas-max ;*
4. *Un tableau trié par ordre croissant (resp. décroissant) vérifie la structure de tas-min (resp.max).*

Proposition 4.1 *La hauteur d'un tas à n éléments est en $O(\log(n))$.*

On fournit l'implémentation de `Inserer(Q, x, k)`, `Extraire_min(Q)` et `Réduire_clé(Q, x, k)`. On montre qu'elles se font en $O(log(n))$.

Proposition 4.2 *Avec une implémentation de tas-min pour la file de priorité et de liste d'adjacence pour le graphe, l'algorithme de Dijkstra est en $O((|V| + |E|) \log(|V|))$.*

Développement 4 : Construction d'un tas en temps linéaire et tri par tas.

Application 4.1 *Deux autres applications des files de priorité :*

- *L'algorithme de Prim ;*
- *L'ordonnancement des taches avec priorité (attention, c'est pas toujours des tas, par exemple l'ordonnanceur linux utilise des arbres rouge noir)*

III Introduction à l'analyse amortie [Cormen et al., 2009]

A Définition

Parfois une simple analyse de la complexité dans le pire cas n'est pas représentative du nombre d'opérations effectuées effectuées par un algorithme. On introduit alors la complexité amortie. Elle permet de majorer le coût total d'une séquence d'opérations et d'associer a chaque opération un coût amorti.

Il existe trois méthodes majeures qui permettent de déterminer une complexité amortie.

Méthode de l'agrégat

Définition 4.7 *On considère une suite de n opérations licite sur une structure de donnée. Soit $T(n)$ le coût total de ces n opérations. Le coût amorti de chaque opération est alors $T(n)/n$.*

Exemple 4.5 *Supposons que l'on effectue une suite de $n-1$ opérations de coût constant puis une opération de coût $O(n)$. Le coût amorti de chacune de ces opérations est $O(1)$.*

Exemple 4.6 *On donne la complexité amortie pour n opérations* `enfiler(F, x)` *et n opérations* `defiler(F)` *sur une file implémentée avec deux listes chaînées en $O(1)$.*

Exercice 4.5 *On suppose que l'on dispose d'un graphe orienté pondéré $G = (V, E, w)$ où $w : E \to \{0, ..., c\}$ où $c \in \mathbb{N}$. Montrer que l'on peut implanter Dijkstra en complexité amortie $\mathcal{O}(|V| + |E|)$ pour ce genre de graphe.*

Méthode du comptable A chaque opération i dans la séquence, on associe un coût fictif f_i qui définira le coût amorti de l'opération. Chaque opération a aussi son coût réel c_i. Au cours de l'exécution, les opérations pour lesquelles $f_i > c_i$ nous font gagner du crédit et les opérations pour lesquelles $f_i \leq c_i$ coûtent du crédit.
Les coûts fictifs que l'on a associés aux opérations sont valides si $\Sigma_{i=0}^{n} f_i - c_i \geq 0$.
Si f_i est constant en fonction de n alors le coût amorti est constant, sinon, il est en $O(f_i)$.

Remarque 4.4 *Dans cette méthode, on fait payer le coût des opérations dont le coût réel est élevé aux opérations dont le coût réel est plus faible.*

Exemple 4.7 *On donne le coût amorti de n insertions dans un tableau dynamique. Application au coût d'une insertion dans un tas.*

Méthode du potentiel On affecte à un état de la structure de donnée D_i après i opérations $o_1, \ldots, o_n$ un potentiel $\Phi(D_i)$. A l'opération o_{i+1} de coût réel c_{i+1} on associe le coût amorti $c_{i+1} + \Phi(D_i) - \Phi(D_{i+1})$. On peut poser $\Phi(D_0) = 0$. On impose, pour majorer le nombre d'opérations réel d'avoir :

$$\Sigma_{i=1}^{n} c_{i+1} + \Phi(D_i) - \Phi(D_{i+1}) \geq \Sigma_{i=1}^{n} c_i$$

Soit :

$$\Phi(D_n) \geq 0$$

Exemple 4.8 *On fait l'analyse amortie d'une pile implémentée par une liste chaînée à laquelle on ajoute l'opération* `Multi_Depiler`.

IV La structure unir et trouver

A Algorithme de Kruskal

Algorithme de Kruskal

Algorithme 4.2 : Algorithme de Kruskal

Données : Un graphe pondéré G.
Résultat : Un arbre couvrant de poids minimal T.
Initialiser T un forêt sans arcs avec les sommets de G ;
tant que *il existe des sommets de G qui n'ont pas été mis dans l'arbre* **faire**
  Choisir e une arête de G de poids minimal qui ne crée pas de cycle dans T ;
  Ajouter e à T ;
retourner T

L'algorithme de Kruskal sélectionne itérativement, de façon gloutonne les arêtes avec les poids les plus faibles et qui ne créent pas de cycle. On construit une forêt qui petit à petit se réunit pour former un seul arbre à la fin de l'exécution de l'algorithme.

Remarque 4.5 *On ré-utilise une structure que l'on a déjà vu dans le passé, la file de priorité pour stocker les arêtes dont les étiquettes sont leur poids.*

Il nous faut une manière efficace de déterminer si l'ajout d'une arête à la forêt T crée un cycle ou non. Pour cela nous allons introduire une nouvelle structure de données : unir et trouver.

B Définition de la structure unir et trouver [Dasgupta et al., 2008]

La structure unir et trouver permet de représenter et travailler avec des partitions d'un ensemble.

Définition 4.8 *La structure unir et trouvée structure un ensemble E de n éléments arrangés en partitions $P_1, \cdots, P_i$ de $E = \sqcup_{i=1}^{n} P_i$ où chaque partie P_i a un représentant r_i. Elle est définie par les opérations* `Unir(E, i, j)` *qui fait l'union ds parties P_i et P_j et choisit un représentant parmi cette nouvelle partie et* `Trouver(E, x)` *qui renvoie le représentant de x dans E. On ajoute aussi l'opération* `Créer_ensemble(E, x)` *qui ajoute l'ensemble $\{x\}$ à la structure si x n'y appartient pas déjà et crée un ensemble singleton dont le représentant est x.*

Exemple 4.9 *En partant d'un ensemble simple $\{3, 9, 18, 33, 208\}$ on crée la structure d'unir et trouver et on effectue quelques opérations unir et trouver.*
On donne l'exemple de l'algorithme de détection de composantes connexes dans un graphe.

Remarque 4.6 *Même dans des textes écrits en français, il ne sera pas rare de lire Union-Find plutôt que unir et trouver. En anglais on parle aussi de structure d'ensemble disjoints.*

C Implémentations naïves avec des listes

Un ensemble est représenté par une liste chaînée avec un pointeur sur la tête de liste et un sur la queue de liste.

Premier cas : chaque élément pointe directement sur son représentant On obtient une implémentation avec `unir` en $O(n)$ dans le pire cas et `trouver` en $O(1)$.

Second cas : lors de l'union, on ne modifie pas le représentant On obtient une implémentation avec `unir` en $O(1)$ et `trouver` en $O(n)$ dans le pire cas.
On voit apparaître une structure arborescente.

Troisième cas : chaque élément pointe directement sur son représentant et lors de l'union on fait pointer le plus petit ensemble sur le reste On obtient une implémentation avec un coût amorti pour m opérations `Créer_ensemble(E, x)`, `Unir(E, i, j)` et `Trouver(E, x)` dont n opérations `Créer_ensemble(E, x)` en $O(m + n\log(n))$.

D Implémentation avec des forêts

Un ensemble est représenté par un arbre, chaque noeud est un élément et il pointe vers son parent. La racine de l'arbre est le représentant de l'ensemble.

On ajoute des heuristiques pour améliorer la complexité des opérations : l'union par rang et la compression de chemins.[Dasgupta et al., 2008, p. 132]

Proposition 4.3 *Pour m opérations trouver sur une structure à n éléments, on obtient un coût amorti en $O((m+n)\log^*(n))$.*

Remarque 4.7 *Soit α la fonction inverse d'Ackermann, en pratique, la complexité amortie est même en $O((m+n)\alpha(n))$ pour m opérations unir et trouver.*

Application 4.2
- *On l'applique à l'algorithme de Kruskal ;*
- *On l'applique au problème de connectivité dynamique ;*
- *On l'applique à l'algorithme de minimisation d'un automate.*

Leçon 5

Implémentations et applications des piles et des files.

Auteur·e·s: Marin Malory, Sorci Émile
Niveau : L1
Pré-requis : Notion d'algorithmiques
Références : [Gaudel et al., 1990], [Cormen et al., 2009]

I Piles et files

Motivation. Les piles et les files sont parmi les structures les plus naturelles et les plus faciles à manipuler. Elles apparaissent dans de nombreux algorithmes, et forment ainsi les structures de données les plus basiques.

A Structure abstraite de pile.

On définit ici la structure abstraite de la pile. On rappelle qu'une structure de donnée abstraite est une liste d'opérations de l'on peut effectuer sur un ensemble de données.

Dans une pile, les insertions et suppressions se sont à une seule extrémité, appelée **sommet de pile**. On appelle aussi cette structure **LIFO** (*Last-In, First-Out*).

Définition 5.1 (Structure abstraite de pile) *Un pile P est une structure de données abstraites ayant les trois opérations :*

- `Empiler(P,x)` *: empile x au sommet de la pile P.*
- `Dépiler(P)` *: dépile P et renvoie l'élément dépilé.*
- `Est_Vide(P)` *: renvoie vrai si la pile P est vide, faux sinon*

On peut aussi ajouter d'autres opérations, comme par exemple `Sommet(P)` qui renvoie le sommet de la pile sans le dépiler.

B Structure abstraite de file.

Dans une file, on insère les éléments en queue de file, et on supprime à la tête (comme dans une file d'attente). On appelle aussi cette structure FIFO (*First-In, First-Out*).

Définition 5.2 (Structure abstraite de pile) *Un file F est une structure de données abstraites ayant les trois opérations :*

- `Enfiler(F,x)` *: enfile x à la queue de la file F.*

— `Défiler(F)` *: défile F et renvoie l'élément qui était en tête de file.*

— `Est_Vide(F)` *: renvoie vrai si la file F est vide, faux sinon*

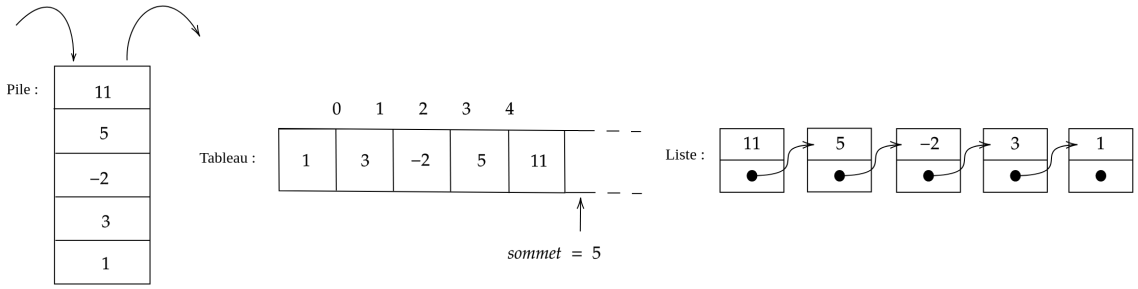
II Implémentations

Plusieurs implémentations pour les piles et les files sont possibles et plus ou moins judicieuses selon le contexte.

Implémentations d'une pile. On présente ici deux implémentations des piles : l'une contiguë et l'autre chaînée.

Contiguë Les éléments de la pile sont rangés dans un tableau `T` et l'on conserve aussi l'indice de sommet de pile dans une variable `som`. Pour dépiler, on renvoie `T[som]` et le sommet de pile devient `som-1`. L'empilement se fait de manière similaire. Il existe deux variantes de cette implémentation, la première qui renvoie une erreur lorsque le tableau est plein, l'autre qui gère les indices de manière circulaire (via des modulos).

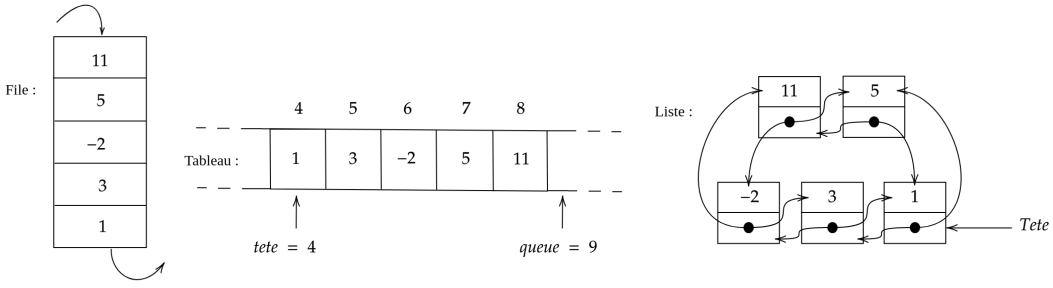
Chaînée Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le premier de la liste ; la pile vide est représentée par `NIL`.



Implémentation d'une file. Comme pour les piles, on présente ici deux implémentations des files, l'une contiguë et l'autre chaînée.

Contiguë Les éléments de la file sont rangés dans un tableau, et on conserve les indices i et j ($i < j$) de début et fin de file.

Chaînée Dans le cas d'une représentation chaînée, soit on a deux pointeurs, `tête` et `queue`, vers le premier et dernier élément de la file ; soit on récupère le premier élément via le pointeur du dernier élément (comme sur la figure).



III Applications des piles et files

A Application des piles.

Parcours en profondeur. On s'intéresse ici aux parcours d'un arbre binaire. On rappelle qu'un parcours consiste à examiner systématiquement, dans un certain ordre, chacun des nœuds de l'arbre pour y effectuer un même traitement.

On présente ici l'algorithme itératif de parcours en profondeur, qui nécessite l'utilisation d'une pile.

> **Remarque 5.1** *Le parcours en profondeur s'écrit plus élégamment en récursif. L'utilisation d'une pile permet souvent de dérécursifier certains algorithmes.*

Calcul de l'enveloppe convexe. Étant donné un ensemble Q de points sur le plan, un problème classique consiste à trouver l'enveloppe convexe $\mathbf{EC}(Q)$, c'est-à-dire le plus petit polygone convexe qui contient l'ensemble des points de Q. Un algorithme classique pour calculer cette enveloppe convexe est le **balayage de Graham**, qui utilise une pile de manière astucieuse.
Développement 1. Présentation du balayage de Graham et correction.

B Application des files.

Parcours en largeur. Un parcours en largeur consiste à explorer un arbre niveau par niveau. Il nécessite l'utilisation d'une file pour stocker les sommets.

Exercice 5.1 *Écrire le pseudo-code du parcours en largeur.*

Algorithme de remplacement de page. Dans le cadre d'une hierarchie mémoire (avec une petite mémoire rapide et une grande mémoire lente), il faut un algorithme de remplacement de page. Deux des algorithmes les plus connus sont FIFO et LRU (*Least-Recently-Used*). La première consiste à remplacer la page en mémoire cache qui est rentrée en première, tandis que la deuxième consiste à remplacer la page qui a été utilisée le moins récemment.
Développement 21. Analyse des performances des algorithmes onlines de remplacement de pages.

> **Remarque 5.2** *L'algorithme FIFO s'implémente directement via une file, ce qui n'est pas le cas de LRU qui nécessite une valeur de priorité pour chaque élément. On aura alors besoin des files de priorités.*

IV Les files de priorité

A Algorithme de Dijkstra [Dasgupta et al., 2008]

Algorithme 5.1 : Dijkstra(G,s)

Données : Un graphe pondéré G et un sommet s
Sorties : Plus courts chemins de s vers les autres sommets
pour $v \in V$ **faire**
  $d[v] \leftarrow +\infty$;
distance[s] $\leftarrow 0$;
$F \leftarrow$ `StructureVide()` ;
Insérer v dans F avec la clé 0 ;
tant que F *non vide* **faire**
  Extraire u de F de clé minimal d_u ;
  pour v *voisins de* u *dans* G **faire**
    $d \leftarrow d_u + w(u,v)$;
    si $d \leq$ *distance[v]* **alors**
      si *distance[v]* $= +\infty$ **alors**
        Insérer v dans F avec la clé d ;
      sinon
        $\#v$ est déjà dans F avec une clé $> d$;
        Diminuer la clé de v dans F à d ;
retourner *distance*

L'algorithme de Dijkstra sélectionne itérativement et de façon gloutonne les sommets les plus proches de s qui sont accessibles par le sommet courant. Il utilise un front de sommets éligibles au cours de son exécution.

Remarque 5.3 *On ne s'intéresse pas ici à la correction de l'algorithme mais aux structures de données à utiliser.*

Besoin de deux structures de données : une pour le graphe et une seconde que l'on appelle file de priorité.

Nous allons évidemment utiliser une implémentation du graphe avec des listes d'adjacence car on a besoin de parcourir les sommets adjacents.

B La structure de donnée

Définition 5.3 *Une* **file de priorité** *est une structure qui permet d'organiser un ensemble d'éléments munis de clés, l'ensemble de clés étant ordonné. Une file de priorité* `Q` *est définie pour tout élément* `x` *associé à une clé* `k` *par les opérations* `Inserer(Q, x, k)`, `Extraire_min(Q)`, `Est_vide(Q)` *et* `Réduire_clé(Q, x, k)`.

C Implémentation via un tas

Définition 5.4 *Un* **tas-min** *(resp. max) est un arbre binaire complet gauche dont les noeuds sont étiquetés par des éléments munis de clés qui vérifie la propriété du tas-min : la clé d'un nœud est inférieure aux clés de ses enfants.*

Exemple 5.1 *On donne un tas-max pour la liste [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]. On donne un tas-min avec ces éléments ainsi que plusieurs exemples d'arbres qui ne sont pas des tas.*

On utilise un tableau indexé de 1 à n pour stocker un tas de n éléments. On accède aux enfants d'un noeud d'indice i en prennant les éléments d'indice $2i$ et $2i+1$ et on accède au parent en prenant l'élément d'indice $\lfloor i/2 \rfloor$.

Exemple 5.2

1. $[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ *vérifie la structure de tas-max ;*
2. *On montre qu'il s'agit du parcours en largeur de l'arbre.*
3. $[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]$ *ne vérifie pas la structure de tas-max ;*
4. *Un tableau trié par ordre croissant (resp. décroissant) vérifie la structure de tas-min (resp.max).*

Proposition 5.1 *La hauteur d'un tas à n éléments est en $O(\log(n))$.*

On fournit l'implémentation de `Inserer(Q, x, k)`, `Extraire_min(Q)` et `Réduire_clé(Q, x, k)`. On montre qu'elles se font en $O(log(n))$.

Proposition 5.2 *Avec une implémentation de tas-min pour la file de priorité et de liste d'adjacence pour le graphe, l'algorithme de Dijkstra est en $O((|V| + |E|) \log(|V|))$.*

Application 5.1 *Deux autres applications des files de priorité :*

— *L'algorithme de Prim ;*
— *L'ordonnancement des taches avec priorité (attention, c'est pas toujours des tas, par exemple l'ordonnanceur linux utilise des arbres rouge noir)*

V Introduction à la complexité amortie

Dans une **analyse amortie**, on compte la complexité d'une séquence d'opérations. Pour présenter les différentes méthodes d'analyse amortie, on ajoute à notre pile une opération `MULTIDEP`, définie de la manière suivante :

Algorithme 5.2 : MULTIDEP(P,k)

tant que $\neg$ *PILE_VIDE*(P) $\wedge k \neq 0$ **faire**
  DEPILER(P) ;
  $k \leftarrow k - 1$;

Cette fonction dépile les k sommets de la pile et coûte $\min(|P|, k)$ opérations élémentaires.

Méthode de l'agrégat. Dans cette méthode, on montre que pour tout n, une suite de n opérations prend le temps total $T(n)$ dans le cas le plus défavorable. Ainsi, le coût amortie d'une opération sera $T(n)/n$.

On considère n opérations sur notre pile, chacune de ces opérations pouvant être soit `DEPILER`, `EMPILER` ou `MULTIDEP`.

— Via une analyse dans le pire des cas, chaque opération coûte au pire $\mathcal{O}(n)$.
— Via la méthode de l'agrégat, on obtient une complexité amortie en $\mathcal{O}(1)$.

Méthode du comptable. Dans cette méthode, on affecte des coût différents à des opérations différentes (un **crédit**). Le coût attribué à cette opération est alors son coût amorti.

Remarque 5.4 *Ici, la valeur de $T(n)$ ne change pas par rapport à la méthode de l'agrégat, mais on a plus de précision sur le coût amorti de chaque opération.*

Ainsi, en notant c_i le coût de la $i$ème opération, et $\hat{c}_i$ son coût amorti, il faut que :

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

En revenant à notre exemple,

Opérations	coût réel	coût amorti
`EMPILER`	1	2
`DEPILER`	1	0
`MULTIDEP`	$\min(\|P\|, s)$	0

On peut alors montrer par récurrence l'inégalité précédente.

Méthode du potentiel. Le but est ici de représenter le travail prépayé comme une « énergie potentielle ».

On note D_i notre structure après i opérations. On appelle fonction potentiel $\Phi : \{D_i\} \to \mathbb{R}$, tel que $\Phi(D_i)$ est le potentiel de la structure D_i. Le coût amorti est alors :

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Proposition 5.3 *$\sum_{i=1}^{n} \hat{c}_i$ est un majorant du coût réel total si, et seulement si, $\Phi(D_n) \geq \Phi(D_0)$.*

En revenant à notre pile, on pose $\Phi(P_i) = |P_i|$, c'est-à-dire la hauteur de la pile. On retombe alors sur les coûts amorties de la méthode comptable.

Dijkstra en temps linéaire. On regarde ici une application de l'analyse amortie aux files de priorités.

Exercice 5.2 *Montrer que, étant donné un graphe pondéré $G = (V, E, w)$ où $w : E \to \{0, ..., K\}$ avec K une constante, on peut résoudre le problème des plus courts chemins depuis un sommet en temps $\mathcal{O}(|V| + |E|)$.*

Leçon 6

Implémentations et applications des ensembles et des dictionnaires.

Auteur·e·s: Rousseau Guillaume
Niveau : L1/L2
Pré-requis : Bases de structures de données, arbres
Références : [Cormen et al., 2009], [Gaudel et al., 1990]

Introduction

Cette leçon présente deux structures de données abstraites fondamentales : les ensembles et les dictionnaires. Ces deux structures sont très fréquentes, pratiques à utiliser et faciles à implémenter. On étudie de près les dictionnaires, en donnant 3 familles d'implémentations (liste chainée, table de hachage et arbre), et l'on montre comment adapter facilement ces méthodes aux ensembles.

I Structure de donnée abstraite

On donne les spécifications formelles des dictionnaires et des ensembles.

Définition 6.1 (Dictionnaire) *Soit U un ensemble d'éléments appelés valeurs, et K un ensemble d'éléments appelés clés. Un dictionnaire D est une structure de données abstraites ayant les trois opérations :*

- `Inserer(D,k,x)` *: insère le couple (k, x) dans D, en écrasant un éventuel couple (k, x') pré-existant.*
- `Recherche(D,k)` *: renvoie la valeur de x telle que (k, x) est dans D, si elle existe.*
- `Supprime(D,k)` *supprime un éventuel couple (k, x) présent dans D.*

On appelle également parfois les dictionnaires des *tableaux associatifs*. Certains langages de programmation utilisent également la terminologie *map*. Dans la suite, on se restreint au cas où $K = \mathbb{N}$, ce qui n'est pas contraignant, car en pratique on manipule des objets dans un ensemble discret, codés en binaire.

Définition 6.2 (Ensemble) *Soit U un ensemble d'éléments. Un* **ensemble** *S est une structure de donnée abstraite défini par les opérations suivantes :*

- `Inserer(S,x)` *: insère x dans S.*
- `Recherche(S,x)` *: renvoie vrai si, et seulement si x est dans S.*
- `Supprime(S,x)` *supprime x de S s'il est présent.*

On remarque la similarité entre les ensembles et les dictionnaires. Nous allons supposer que tous les éléments $x \in U$ ont un attribut de clé primaire $x.cle$, tel que deux éléments de U distincts auront des clés distinctes. Alors, on peut implémenter directement un ensemble à l'aide d'un dictionnaire.

Exercice 6.1 *Donner une implémentation des ensembles utilisant les dictionnaires.*

Cette hypothèse sur les éléments de U n'est pas si contraignante, car en pratique on considère toujours des éléments ayant une clé identifiable. En Python, la fonction `id` renvoie pour chaque objet une valeur distincte et constante tout au long de l'exécution du programme, et peut donc simuler une clé. En C, l'adresse mémoire peut directement servir de clé, car deux objets du même type n'auront jamais la même adresse.

II Premières implémentations

Nous allons étudier quelques manières d'implémenter les dictionnaires. Dans chaque version, on s'intéresse à la complexité des trois primitives. Plus précisément, on regarde la complexité dans le pire cas, et la complexité en moyenne. On distingue également les complexités des recherches fructueuses et infructueuses. Nous verrons au fur et à mesure le modèle aléatoire considéré.

Listes chainées Voyons une implémentation naïve des dictionnaires, par liste chaînée. Dans cette implémentation, un dictionnaire D est une liste chaînée de couples (k, x).

— Insertion : on insère les nouveaux couples en tête de liste.

— Recherche : on recherche linéairement dans la liste.

— Suppression : on recherche linéairement, puis on supprime le maillon, en redirigeant les pointeurs si besoin.

Proposition 6.1 *Dans une implémentation d'un dictionnaire via une liste chaînée :*

— *l'insertion se fait en $\mathcal{O}(1)$ opérations élémentaires ;*

— *la recherche et la suppression se fait en $\mathcal{O}(n)$ opérations élémentaires où n est le nombre de clés dans le dictionnaire, dans le pire des cas et en moyenne.*

Tableaux à adressage direct. On suppose maintenant que les clés sont non seulement des entiers mais sont en plus bornées : $K = [0, m-1]$ pour un certain $m \in \mathbf{N}$. Alors, on peut implémenter notre dictionnaire directement avec un tableau T de taille m :

— Insertion : Pour insérer le couple (k, x), on effectue l'opération $T[k] \leftarrow x$

— Recherche : Pour rechercher la valeur de k, on renvoie $T[k]$, si cette case est non vide.

— Suppression : Pour supprimer la valeur de k, on vide $T[k]$.

Toutes ces opérations sont en $O(1)$! En revanche, il faut allouer au démarrage un espace mémoire proportionnel à la taille de l'univers des clés, y compris lorsque l'on utilise très peu de clés.

Exemple 6.1 *On souhaite stocker des informations concernant les variables lors d'une étape de compilation. En autorisant les 26 lettres minuscules et majuscules, et en bornant la taille du nom des variables par 40, notre tableau doit contenir au moins 26^{40} cases : ce n'est pas réalisable !*

III Arbres de recherches

A Arbre binaire de recherche

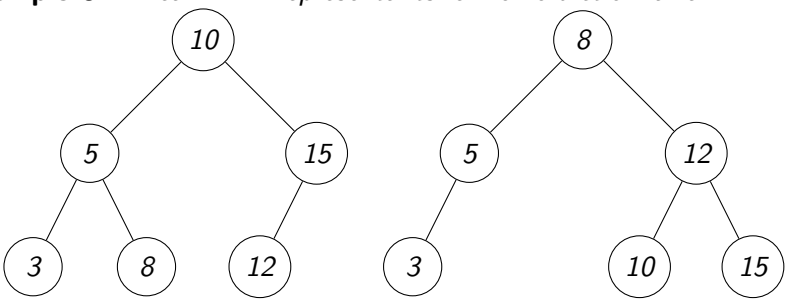
Voyons une dernière manière d'implémenter les dictionnaires, en utilisant des arbres. L'idée générale est de stocker les couples (k, x) dans un arbre, de telle sorte qu'il soit facile de retrouver un élément en partant de la racine.

Définition 6.3 (Arbre binaire de recherche) *Soit A un arbre binaire enraciné, étiqueté par une fonction $e : A \to U$. On dit que A est un* **arbre binaire de recherche** *(ou* **ABR***) si pour tout nœud n, en notant g et d ses fils gauche et droit (s'ils existent), on a $e(g) < e(n) < e(d)$.*

De tels ABR ne peuvent que stocker des clés, mais on modifie facilement la structure pour pouvoir rajouter des données satellites sur les nœuds, et donc implémenter les dictionnaires. A nouveau, cette structure nécessite que l'ensemble où les clés vivent soit muni d'un ordre total.

Plusieurs ABR distincts peuvent représenter le même dictionnaire.

Exemple 6.2 *Deux ABR représentants le même dictionnaire.*



Recherche et insertion. Dans un ABR, on insère un élément x en descendant depuis la racine, en prenant le fils gauche ou le fils droit selon si x est plus petit ou plus grand que la racine courante, et en créant un nouveau nœud étiqueté par x lorsque l'on ne peut plus avancer. L'opération de recherche se fait de manière analogue.

Exercice 6.2 *Définir un type* `tree` *en OCaml où chaque nœud contient un entier. Écrire les fonctions* `recherche : tree -> bool` *et* `insert : tree -> int -> tree`.

Suppression d'un élément. La suppression est un peu plus subtil. Lorsque le nœud est une feuille ou si le nœud n'a qu'un enfant, alors la transformation est simple. Par contre, si le nœud possède deux enfants, alors il faut retirer le minimum du sous-arbre droit (ou le maximum du sous-arbre gauche) pour le remplacer.

Exercice 6.3 *Écrire une fonction* `extraire_min : tree -> (tree*int)` *qui, étant donné un ABR, retourne le couple correspondant au minimum et à l'arbre obtenu en le supprimant. En déduire une fonction* `supprime : tree -> int -> tree`.

— Arbre parfaitement équilibré : soit A un ABR contenant n valeurs, tel que chaque feuille est à la même hauteur h. Alors, $h = O(\log(n))$, et donc l'insertion, la recherche se font en $(O(\log(n))$.

— Arbre linéaire : Soit A un ABR contenant n valeurs, tel qu'aucun nœud n'a de fils gauche. Alors, sa hauteur est $h = n$: L'insertion et la recherche se font en $O(n)$.

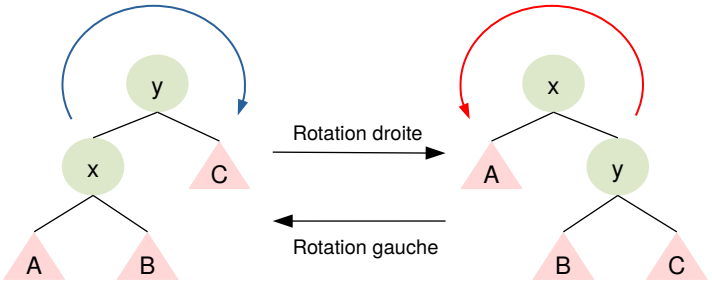
Complexité. Cet exemple souligne un problème majeur de l'implémentation par ABR : l'ordre d'insertion des éléments détermine le squelette de l'arbre. La hauteur peut alors être linéaire.

Proposition 6.2 *Soit A un ABR à n nœuds et de hauteur h. La recherche, l'insertion et la suppression se font dans le pire cas en $\mathcal{O}(h)$ comparaisons.*

B Arbres équilibrés.

Arbres AVL. Une solution est de rééquilibrer l'arbre de recherche au fil des insertions, et de garantir à chaque instant une hauteur $h = O(\log(n))$. Une telle solution a été notamment implémenté dans les **AVL** (Adelson-Velsky et Landis) dont les mécanismes de rééquilibrages se basent sur des opérations de rotation d'arbres. Un tel arbre est dit automatiquement équilibré.

Exemple 6.3



B-arbres. Dans certains cas, le côté ≪ binaire ≫ des arbres de recherches peuvent poser problème. En effet, regardons le cas d'une hiérarchie mémoire où les données sont stockés dans une mémoire lente (comme le disque par exemple). Au lieu de rapatrier un nœud après l'autre, il peut être plus intéressant de créer de plus ≪ gros nœuds ≫.
Développement 3. B-arbres.

IV Tables de hachage

Dans la méthode arborescente vu précédemment, il y a un ordre sur les clés, et la place de la clé dépend du rang de sa clé par rapport aux clés des autres éléments. Cependant, dans les méthodes de hachage la place d'un élément est calculée uniquement à partir de sa clé.

Exemple 6.4 *Si $K = \mathbb{N}$ et qu'on veut utiliser un tableau de taille $m << |K|$ fixée, on peut discriminer $k \in K$ par $(k \mod m)$.*

L'objet fondamental des tables de hachage est la **fonction de hachage**, qui transforme la clé en une adresse dans une zone de mémoire contiguë ou un indice de tableau. Les cases de ce ≪ petit ≫ tableau sont appelées **alvéoles**.

Définition 6.4 (Fonction de hachage) *Soit K un ensemble de clés, et $m \in \mathbb{N}$ avec $m << |U|$. Une fonction de hachage est une fonction $h : K \to [0, m-1]$.*

Donc, lorsque l'on souhaite stocker le couple (k, x), au lieu de stocker x dans la case k, on calcule $h(k)$, et l'on stocke x dans cette case. Comme h n'est pas injective ($< |K|$), il existe des valeurs $k \neq k'$ telles que $h(k) = h(k')$. On appelle **collision** un tel couple, et on dit qu'il y a collision dans une table de hachage lorsque l'on insère une valeur dont la clé a le même hash qu'une clé déjà présente dans la table.

Exemple 6.5 *Si $K = \Sigma^{\leq m-1}$ un ensemble de chaînes de caractères de taille au plus $m-1$ sur un alphabet Σ. Une fonction de hachage possible consiste à associer à chaque mot sa taille. Il y a une collision pour chaque mot de même longueur.*

Choix de la fonction de hachage. Dans le premier exemple, la fonction de hachage est un reste modulo m. Lorsque l'on prend $m = 2^p$ une puissance de 2, le hash d'une clé est constitué des p premier bits de cette clé. En pratique, il peut être souhaitable que la fonction de hachage ne reflète pas la structure des clés, et donc le choix de m est important.

En particulier, il est souhaitable le hachage soit uniforme. Lorsque

$$\forall x \in U, \forall i \in \{0, ..., m-1\}, Pr[h(x) = i] = \frac{1}{m}$$

on parle de **hachage uniforme simple**.

Il y a deux manières de gérer les collisions :

— par **chaînage** : chaque case du tableau contient une liste chaînée, et donc si deux clés on le même hash, elles se cumulent dans la même liste chaînée.

— par **adressage ouvert** : lorsque l'alvéole est déjà prise, on en cherche une autre.

Analyse de complexité. Quelle est l'efficacité du hachage avec chaînage ? En particulier, combien de temps faut-il pour rechercher un élément dont on connaît la clé ?

Étant donnée une table de hachage T à m alvéoles qui stocke n éléments, on définit pour T le facteur de remplissage α par n/m, c'est-à-dire le nombre moyen d'éléments stockés dans une chaîne. Notre analyse se fera en fonction de α.

Théorème 6.1 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche infructueuse prend un temps moyen $\Theta(1+\alpha)$, sous l'hypothèse d'un hachage uniforme simple.*

Théorème 6.2 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche réussie prend en moyenne un temps $\Theta(1+\alpha)$, sous l'hypothèse d'un hachage uniforme simple.*

Développement 5. Démonstrations des théorèmes précédents.

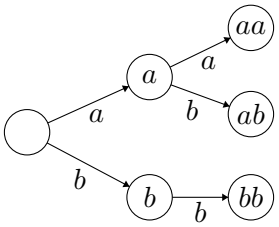
V Ensemble de mots

Les tables de hachage et les ABR permettent de stocker des données très générales. En contrepartie, la structure des données que l'on cherche à stocker n'est pas exploitée. Dans certains cas, cette structure peut mener à des implémentations encore plus efficaces que les tables de hachage. Intéressons nous au problème suivant.

On cherche à stocker un ensemble de mots sur l'alphabet Σ, de façon à pouvoir ajouter, chercher et supprimer des mots de cet ensemble. On souhaite également stocker, pour chaque mot, des données satellites (par exemple : une définition pour les mots du dictionnaire d'une langue).

Les *arbres préfixes* (ou *tries*) sont une excellente manière d'implémenter de tels ensembles. Voyons sur un exemple comment ces arbres fonctionnent :

Exemple 6.6 *On considère l'arbre préfixe pour l'ensemble des mots a, aa, ab et bb.*



Les arbres préfixes sont utilisés par exemple pour les algorithmes de prédiction de texte. Ils ont l'avantage d'être très efficaces en espace et en temps : pour stocker des mots $w_1, \ldots, w_n$, un arbre préfixe utilise un espace mémoire $\mathcal{O}(\sum |w_i|)$, et la recherche ou l'insertion d'un mot w se font en $\mathcal{O}(|w|)$. En particulier, c'est indépendant du nombre d'éléments présents !

Leçon 7

Accessibilité et chemins dans un graphe. Applications.

Auteur·e·s: Marin Malory
Niveau : MPI
Pré-requis : Théorie des graphes basique, représentation des graphes et parcours
Références : [Gaudel et al., 1990], [Cormen et al., 2009],[Benoit et al., 2013]

I Terminologie

Définition 7.1 (Chemin) *Dans un graphe orienté G (resp. non orienté), on appelle* **chemin** *de longueur λ, une suite de $(\lambda+1)$ sommets $(s_0, s_1, ..., s_\lambda)$ tel que : pour tout $0 \leq i \leq \lambda - 1$, (s_i, s_{i+1}) est un arc (resp. une arête) de G.*

Remarque 7.1 *Par convention, on dit qu'il y a un chemin de longueur 0 de tout sommet vers lui-même. Dans un graphe non-orienté, les chemins sont aussi appelés* **chaînes**.

Définition 7.2 (Chemin élémentaire) *Un chemin est dit* **élémentaire** *s'il ne contient pas plusieurs fois le même sommet.*

Exercice 7.1 *Montrer que deux chemins élémentaires de longueur maximale dans un graphe connexe G ont un sommet en commun.*

Définition 7.3 (Circuit/Cycle) *Dans un graphe orienté (resp. non orienté), un chemin $(s_0, ..., s_\lambda)$ dont les λ arc (resp. arêtes) sont distincts deux à deux et tels que $s_0 = s_\lambda$, est un circuit (resp. un cycle).*

Exercice 7.2 *On dit qu'un graphe non orienté est Eulérien s'il existe un cycle passant par chaque arête exactement une fois. Montrer qu'un graphe est eulérien si et seulement si tous ses sommets sont de degré pair.*

II Accessibilité

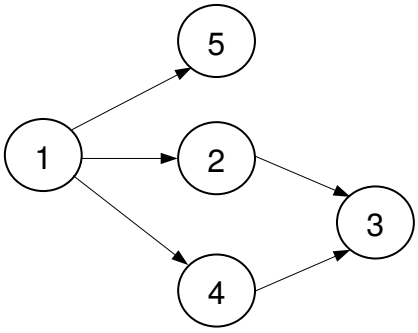
Définition 7.4 (Accessibilité) *Étant donné un graphe G (orienté ou non) et deux sommets s et t, on dit que t est accessible depuis s s'il existe un chemin allant de s à t dans G.*

A Tri topologique

Définition 7.5 (Tri topologique) *Étant donné un graphe orienté acyclique $G = (V, E)$ avec $V = \{v_1, ..., v_n\}$, un tri topologique de V est une permutation $\sigma : [n] \to [n]$ tel que pour tout $(u_i, u_j) \in E$, $\sigma(i) < \sigma(j)$.*

Proposition 7.1 *Tout graphe orienté acyclique admet un tri topologique.*

Exemple 7.1 *On souhaite exécuter 5 tâches $t_1, ..., t_5$ sur un unique processeur. Cependant ces tâches suivent le graphe de dépendance suivant :*



Un tri topologique de ce graphe est alors $(1, 2, 5, 4, 3)$, et on peut exécuter les tâches dans cet ordre sans créer de problème.

Théorème 7.1 *L'algorithme qui, étant donné un graphe orienté acyclique, réalise un parcours en profondeur où la fonction de Post-traitement(u) ajoute u à une liste et retourne cette liste réalise un tri topologique.*

B Composantes connexes et fortement connexes

Connexité.

Définition 7.6 (Connexité) *Un graphe orienté est dit* **fortement connexe** *si pour tout couple de sommets distincts (u, v), u est accessible depuis v et v depuis u.*

Un graphe non orienté est dit **connexe** *si pour toute pair de sommets $\{u, v\}$, u est accessible depuis v.*

À partir du parcours en profondeur, on peut aussi calculer les composantes connexes d'un graphe non orienté facilement. En effet, on a exactement une composante connexe par appel à la fonction Visiter-PP.

Composantes fortement connexes

Définition 7.7 (Composante fortement connexe (CFC)) *On appelle* **composante fortement connexe** *d'un graphe orienté un sous-graphe fortement connexe maximal, c'est-à-dire un sous-graphe fortement connexe qui n'est pas strictement contenu dans un autre sous-graphe fortement connexe.*

Exercice 7.3 *Montrer que le graphe des composantes fortement connexes, obtenu en remplaçant chaque CFC par un unique sommet, est un graphe orienté acyclique.*

Le calcul des composantes fortement connexes est plus compliqué qu'un simple parcours.

Théorème 7.2 *Étant donné un graphe orienté $G = (V, E)$, on peut calculer les composantes fortement de connexes en temps linéaire, $\mathcal{O}(|V| + |E|)$.*

Deux algorithmes permettant de prouver ce théorème :

— l'algorithme de Kosaraju (deux parcours en profondeur, l'un dans G^T et l'autre dans G) ;

— l'algorithme de Tarjan (un unique parcours en profondeur).

Développement 14. Ces algorithmes permettent de résoudre le problème 2-$\mathbf{SAT}$ en temps linéaire.

Définition 7.8 (2-SAT) *On définit le problème 2-$\mathbf{SAT}$ de la manière suivante :*

— **Données** *: n variables booléennes $v_1, ..., v_n$ et m clauses de la forme $x \vee y$ où x et y sont des variables ou leur négation.*

— **Question** *: existe-t-il une affectation $\mu : \{v_1, ..., v_n\} \to \{0, 1\}$ tel que pour toute clause c, $\mu(c) = 1$.*

Théorème 7.3 *2-$\mathbf{SAT}$ peut être résolu en temps linéaire.*

III Plus court chemin dans un graphe pondéré

Définition 7.9 (Graphe pondéré) *Un graphe pondéré est un graphe $G = (V, E)$ muni d'une fonction de poids $w : E \to \mathbb{R}$. On peut étendre w à V^2 en posant $w(u, v) = +\infty$ lorsque $(u, v) \notin E$.*
Le poids d'un chemin est alors défini comme la somme des poids des arêtes qui le compose.

Définition 7.10 (Distance dans un graphe pondéré) *Étant donné un graphe pondéré (G, w) dans circuit de poids strictement négatif, on définit la distance de u à v par :*

$$d(u, v) = \min\{w(c) | c \text{ est un chemin reliant } u \text{ à } v\}$$

A Algorithme de Floyd-Warshall

On essaye ici de trouver les plus courts chemins pour toutes les paires de sommets. On supposera $G = (V, E, w)$ avec $V = \{0, ..., n-1\}$, et si $ij \notin E$, alors on pose $w(ij) = +\infty$.

Cet algorithme utilise la programmation dynamique : en notant $d_{ij}^{(k)}$ la distance du plus court chemin allant de i à j passant par les k premiers sommets, on a

$$\begin{cases} d_{ij}^{(k)} = w_{ij} & \text{si } k = 0 \\ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1 \end{cases}$$

L'algorithme de Floyd-Warshall implémente directement cette idée avec une complexité temporelle $\mathcal{O}(|V|^3)$.

Exercice 7.4 (Plus grande bande passante) *On définit la bande passante d'un chemin comme le plus petit poids d'une arête sur ce chemin. Proposer un algorithme qui, pour toute paire de sommet, détermine la plus grande bande passante d'un chemin entre ces deux sommets.*

B Algorithme de Dijkstra

On essaye ici, étant donné une source s, de trouver l'ensemble des plus courts chemin de s aux autres sommets.

Algorithme 7.1 : Dijkstra(G,s)

```
pour u ∈ V faire
    distance[v] ← +∞ ;
distance[s] ← 0 ;
F ← FilePriorité() ;
Insérer(F, s, 0) ;
tant que ¬ estVide(F) faire
    u, d_u ← ExtraireMin(F) ;
    pour v ∈ N(u) faire
        d ← d_u + w(u, v) ;
        si d < distance[v] alors
            si distance[v] = +∞ alors
                Insérer(F,v,d) ;
            sinon
                DiminuerPriorité(F,v,d) ;
            distance[v] ← d ;
retourner distance
```

Théorème 7.4 *L'algorithme de Dikstra est correct.*

Théorème 7.5 *L'algorithme de Dijkstra réalise au plus $|V|$ appels à Insérer et ExtraireMin, et $|E|$ appels à DiminuerPriorité.*

Implémentation. Une structure de file de priorité doit être implémenter pour l'algorithme de Dijkstra. Par exemple, on utilisant un tas-min, on obtient les opération élémentaires en $\mathcal{O}(\log n)$ où n est le nombre d'éléments dans le tas. Ainsi, la complexité de Dijsktra est en $\mathcal{O}((|V| + |E|) \log |V|)$.

Exercice 7.5 *En supposant que la fonction de poids vérifie $w : E \to \{0, ..., K\}$, montrer que l'on peut implémenter l'algorithme de Dijkstra en temps linéaire.*

IV Exemples de problèmes NP-complets

A Le problème du voyageur de commerce

On considère un graphe $G = (V, E)$. Un cycle hamiltonien d'un tel graphe est un cycle passant une et une unique fois par chaque sommet. Déterminer si un graphe admet un cycle hamiltonien s'avère être **NP**-complet.

On considère ici un problème proche : le voyageur de commerce.

Définition 7.11 (Problème du voyageur de commerce (TSP)) *Étant donné un graphe pondéré complet $G = (V, E)$, une fonction de coût $w : E \to \mathbf{N}$, existe-t-il un cycle C passant par chaque sommet une et une seule fois avec $\sum_{e \in C} w(e) \leq k$?*

Théorème 7.6 *Le problème* **TSP** *est* **NP***-complet.*

On a définit ci-dessus un problème de décision, on nommera de la même manière le problème d'optimisation qui en découle.

Théorème 7.7 *Pour tout $\lambda \geq 1$, il n'existe pas de λ-approximation de* **TSP**, *à moins que* $\mathbf{P} = \mathbf{NP}$.

On peut tout de même restreindre le problème au cas où la fonction de coût w vérifie l'inégalité triangulaire. On obtient le problème $\mathbf{TSP} - \mathbf{EUCLIDE}$.

Théorème 7.8 *Il existe une 2-approximation du problème* $\mathbf{TSP} - \mathbf{EUCLIDE}$.

Développement 13. Preuve des trois théorèmes précédents.

B Chemin disjoints

On considère le problème d'optimisation suivant, consistant à réserver des chemins disjoints dans un graphe pour satisfaire des requêtes.

Définition 7.12 (Maximum-Edge Disjoint Path (MEDP))

- **Données** *: un graphe $G = (V, E)$ et un ensemble de requêtes $R \subset V \times V$.*
- **Sortie** *: nombre maximal de requêtes réalisables, en sachant que cet ensemble est réalisable s'il existe un ensemble de chemins deux à deux à arêtes disjointes qui réalise les requêtes.*

Ce problème s'avère être **NP**-complet aussi, mais on peut trouver des approximations.

Théorème 7.9 *Il existe une $(n-1)$-approximation pour le problème* **MEDP**, *où n est le nombre de sommets du graphe en entrée.*

Théorème 7.10 *Il existe une $\lceil \sqrt{m} \rceil$-approximation pour le problème* $\mathbf{MEDP}$*, où m est le nombre d'arêtes du graphe en entrée.*

On peut aussi montrer que l'algorithme précédent est le meilleur possible : il n'existe aucune $m^{\frac{1}{2}-\epsilon}$-approximation ($\epsilon > 0$) à moins que $\mathbf{P} = \mathbf{NP}$.

Leçon 8

Algorithmes de tri. Exemples, complexité et applications.

Auteur·e·s: Marin Malory
Niveau : L2-L3
Pré-requis : Algorithmique, Structures de données et Complexité
Références : [Gaudel et al., 1990], [Cormen et al., 2009],
[Benoit et al., 2013], [Goodrich and Tamassia, 2014]

I Introduction et méthodes simples

A Introduction à l'étude des méthodes de tri

Les méthodes de tri sont très importantes en pratique, en particulier en informatique de gestion où beaucoup d'applications consistent à trier les fichiers.

Spécification.

Définition 8.1 (Problème de tri) *Étant donné un tableau de n éléments $T = [e_1, ..., e_n]$ où chaque élément e_i est muni d'une clé $c(e_i) \in U$ où U est totalement ordonné, le résultat du problème du tri est une permutation de L, noté $L' = [e'_1, ..., e'_n]$ où pour tout $1 \leq i < j \leq n$, $c(e'_i) \leq c(e'_j)$.*

Exemple 8.1 *On cherche à trier les mots [aaa,bb,aaaa,c] où la clé de chaque mot est sa taille. La solution au problème du tri est alors la liste [c,bb,aaa,aaaa].*

Stabilité. Dans le cas où les éléments de la liste à trier ont une même clé, la solution au problème n'est pas unique. On dit qu'une méthode de tri est **stable** si en cas d'égalité de clés, les éléments triés conserve l'ordre de départ.

Organisation de la mémoire. Si la liste à trier ne tient pas en mémoire, on est amené à utiliser des techniques particulières, dites **tris externes**. À l'inverse, si on peut tout faire en mémoire centrale, on réalise un **tri interne**. On dit qu'un tri est **en place** si on ne réalise pas de copie de la liste à trier.

Convention. Pour faciliter la compréhension, on se contente dans cette leçon de trier des listes d'entiers, et on se concentrera sur les tris internes.

B Méthodes par sélection

Sélection ordinaire. L'idée est de séquentiellement récupérer le minimum. Ce tri est en place et stable.

Algorithme 8.1 : TriSélection(L)

```
n ← |T| ;
pour i = 0...n − 1 faire
    i_min ← argmin_{i≤j≤n−1} T[j] ;
    T[i] ↔ T[i_min] ;
```

Exercice 8.1 *En se rappelant que le nombre de comparaisons nécessaires pour récupérer le minimum d'une liste k éléments est $k-1$, montrer que TriSélection réalise exactement $\frac{n(n-1)}{2}$ comparaisons et $n-1$ échanges.*

Tri à bulles. Voici ici l'exemple d'un algorithme peu efficace mais simple à implémenter. L'idée est de parcourir la liste par la fin, et à chaque fois qu'on rencontre deux éléments successifs non triés, on les échange.

Algorithme 8.2 : TriBulles(L)

```
n ← |T| ;
i ← 0 ;
tant que i < n faire
    pour j = n − 1...i + 1 faire
        si T[j] < T[j − 1] alors
            T[j] ↔ T[j + 1] ;
```

Exercice 8.2 *Montrer que le nombre de comparaisons de TriBulles est le même que celui de TriSélection. En déduire une borne sur le nombre d'échanges.*

Théorème 8.1 *En supposant que la liste L est une permutation de $\{1,...,n\}$ tiré uniformément, le nombre moyen d'échange de TriBulles(L) est un $\mathcal{O}(n^2)$.*

C Méthodes par insertion

Insertions séquentielles. L'idée est de maintenir un tableau trié, et d'insérer au fur et à mesure les éléments de notre tableau d'origine. Cet algorithme est naturellement récursif.

Algorithme 8.3 : TriInsertion(T, i)

```
si i > 1 alors
    TriInsertion(T, i − 1) ; # trie du début de la liste
    k ← i − 1 ;
    x ← T[i] ;
    tant que k ≥ 0 et T[k] > x faire
        T[k + 1] ← T[k] ;
        k ← k − 1 ;
    T[k + 1] ← x ;
```

Exercice 8.3 (Tri d'une liste en Ocaml) *Écrire un fonction OCaml* `insert : int -> int list -> int list` *qui insère un élément dans une liste triée, et une fonction* `triInsertion : int list -> int list` *réalisant un tri insertion d'une liste d'entier.*

Théorème 8.2 (Correction et complexité du tri insertion) *Étant donné un tableau T à n éléments, la procédure TriInsertion$(T, n-1)$ trie T en réalisant dans le pire cas $\frac{n(n-1)}{2} - 1$ comparaisons.*

Théorème 8.3 (Complexité en moyenne du tri insertion) *Étant donné un tableau à n éléments, l'algorithme de tri par insertion réalise en moyenne $\frac{n^2+3n}{4} - 1$ comparaisons.*

Remarque 8.1 *Au lieu de compter le nombre de comparaisons, on pourrait compter le nombre de* **transfert** *(un transfert est une opération de la forme $T[...] \leftarrow ...$). Ce nombre reste quadratique en la taille de la liste.*

Le tri par insertions séquentielles n'utilise pas les spécificités des tableaux, et peut donc être implémenter pour trier une liste comme sur l'exercice précédent.

Puisque l'accès à un élément d'un tableau est en coût constant, on peut rechercher la place de l'élément à insérer en réalisant un nombre logarithmique de comparaisons via dichotomie. On n'améliore pas le nombre de transfert mais le nombre de comparaisons devient un $\mathcal{O}(n \log n)$.

II Algorithmes de tri efficaces

On a vu principalement des tris dont les complexités, en nombre de transfert ou en nombre de comparaisons sont en $\mathcal{O}(n^2)$. On va étudier ici des tries permettant d'atteindre un $\mathcal{O}(n \log n)$.

A Tri fusion

L'algorithme du **tri par fusion** suit fidèlement la méthodologie diviser-pour-régner. Intuitivement, il agit de la manière suivante :

Diviser Diviser la suite de n éléments à trier en deux sous-suites de $n/2$ éléments chacune.

Régner : trier les deux sous-suites de manière récursive en utilisant le tri par fusion.

Combiner : Fusionner les deux sous-suites triées pour produire la réponse triée.

La récursivité s'arrête quand la séquence à trier est de longueur 1.

Exercice 8.4 *Écrire trois fonctions OCaml permettant l'implémentation du tri fusion de deux listes d'entiers :* `split : int list -> (int list * int list)`, `merge : int list -> int list -> int list` *et* `merge_sort : int list -> int list`.

Théorème 8.4 *L'algorithme de tri par fusion trie une séquence d'entiers en $\mathcal{O}(n \log n)$.*

B Tri rapide

L'idée est de partager la liste en deux sous-listes, où tous les éléments de la première sont plus petits que ceux de la seconde. On trie alors récursivement ces deux sous-listes. Pour séparer les deux listes, on choisit un **pivot**.

Comment choisir le pivot ? L'idéal serait de prendre la médiane. Cependant, ce serait trop coûteux en temps. Plusieurs moyens de choisir le pivot existe. On en verra deux : on prend le premier élément de la liste ou un élément au hasard. D'autres choix sont possibles, comme la médiane des trois premiers éléments.

Algorithme 8.4 : TriRapide(T, l, r)

```
si l < r alors
    q ← Partition(T, l, r) ;
    TriRapide(T, l, q − 1) ;
    TriRapide(T, q + 1, r) ;
```

Algorithme de tri rapide. On se donne tout d'abord une procédure `Partition(T,p,r)` qui effectue la partition du sous-tableau $T[p...r]$ et retourne l'indice q du pivot.

Partition du tableau. Le but est ici de partitionner le sous-tableau $T[p...r]$ en choisissant pour pivot $T[r]$.

Algorithme 8.5 : Partition(T, p, r)

```
x ← A[t] ;
i ← p − 1 ;
pour j = p...r − 1 faire
    si A[j] ≤ x alors
        i ← i + 1 ;
        A[i] ↔ A[j] ;
A[i + 1] ↔ A[r] ;
retourner i + 1
```

Théorème 8.5 *La procédure* `TriRapide`$(T, 0, |T| - 1)$ *trie le tableau* T*. Son temps d'exécution est dans le cas le plus défavorable un* $\mathcal{O}(n^2)$*.*

Remarque 8.2

1. *Même si sa complexité est moins bonne que le tri-fusion, le tri rapide est en place. Il est donc plus utilisé en pratique.*
2. *Via une analyse probabiliste, on peut montrer que la complexité moyenne du tri rapide est un* $\mathcal{O}(n \log n)$*. Cette preuve est très similaire à la preuve de l'espérance du nombre de comparaison du tri rapide randomisé.*

Tri rapide randomisé [Cormen et al., 2009, 7.3]

Dans cette version du tri rapide, on choisit le pivot de manière aléatoire. La fonction `PartitionBis` fonctionne comme la fonction `Partition`, mais l'indice du pivot est passé en paramètre au lieu de prendre le dernier élément.

Algorithme 8.6 : TriRapideRandomisé(T, l, r)

```
si l < r alors
    pivot ← Random({l, ..., r}) ;
    pivot ← PartitionBis(T, l,r, pivot) ;
    TriRapideRandomisé(T, l, pivot−1) ;
    TriRapideRandomisé(T,pivot+1, r) ;
```

Théorème 8.6 *L'espérance du nombre de comparaisons du tri rapide randomisé d'un ensemble à n éléments est au plus $2nH_n$ où H_n est le terme général de la série harmonique.*

Développement 6. Preuve du théorème 8.6

C Tri par tas

Développement 4. On peut utiliser la structure de tas afin de créer un algorithme de tri ayant une complexité en $\mathcal{O}(n \log n)$. L'idée est de construire un tas avec les n éléments à trier (possible en temps linéaire), et de réaliser n extractions du tas.

D Optimalité des tris par comparaisons

On peut maintenant se poser s'il est possible de trouver un algorithme encore meilleur. Pour obtenir un résultat intéressant, on va se restreindre à la classe $\mathbf{TC}$ des algorithmes de tri qui opèrent uniquement par comparaison deux à deux des clés des éléments à trier. De plus, on fait l'hypothèse que les clés de la liste à trier sont distinctes.

Théorème 8.7 *La complexité, en nombre de comparaisons, aussi bien moyenne qu'au pire, de tout algorithme de la classe $\mathbf{TC}$ est d'ordre supérieur ou égal à $n \log_2 n$.*

Remarque 8.3 *Tous les algorithmes ne sont pas de la classe $\mathbf{TC}$. Certains algorithmes utilisent des propriétés sur les objets à trier, comme le tri par paquet qui fonctionne sur des nombres réels appartenant à un intervalle borné fixé à l'avance. Cet algorithme est en moyenne linéaire.*

III Exemples d'applications du tri

A Algorithmes gloutons

Dans les algorithmes gloutons, le choix glouton consiste parfois à sélectionner le maximum ou le minimum d'un ensemble. Puisque ce choix est répété séquentiellement, un tri en pré-traitement est souvent judicieux. On donne ici deux exemples.

Moyenne géométrique. Étant donné deux ensembles de réels A et B chacun de taille n, on cherche à ordonner A et B de tel manière à maximiser

$$\prod_{i=1}^{n} a_i^{b_i}$$

L'algorithme glouton qui prend les maximums de A et B, les associe ensemble et recommence est optimal. Il suffit alors de trier A et B, ce qui se fait en $\mathcal{O}(n \log n)$

Problème du gymnase. Le but est d'organiser plusieurs évènements dans un gymnase sans que ceux-ci se chevauchent. On cherche à placer un maximum d'évènements, chacun étant caractérisé par une heure de début d_i et de fin f_i. L'algorithme glouton qui trie les évènements par heure de fin croissante, et place les évènements de manière gloutonne est alors optimal.

B Somme de flottants

Un nombre flottant est une paire d'entiers (m, d) représentant le nombre $m \times b^d$ où b vaut 2 ou 10. Dans tous les environnements de programmation, m et d sont limités : chaque opération arithmétique peut alors avoir une erreur d'arrondi.

On introduit alors une valeur $\epsilon < 1$ appelée **précision machine** afin de borner les erreurs en fonction de ϵ. Par exemple, pour l'addition de deux nombres flottants w et y, on peut écrire

$$fl(x+y) = (x+y).(1+\delta_{xy})$$

avec $|\delta_{xy}| \leq \epsilon$. On considère l'algorithme de somme d'une séquence $(x_1, ..., x_n)$ donné ci-dessous :

Algorithme 8.7 : FloatSum$(x_1, ..., x_n)$

$s \leftarrow 0.0$;
pour $i = 1...n$ **faire**
  $s \leftarrow fl(s + x_i)$;
retourner s

Or, si on considère que ϵ est assez petit tel que pour tout flottant x, $\epsilon^2 x$ est négligeable, alors on peut estimer une borne supérieure e_n pour la différence $d_n = |\text{FloatSum}(x_1, ..., x_n) - (x_1 + ... + x_n)|$:

$$e_n = \epsilon \sum_{i=1}^{n} (n - i + 1)x_i$$

Pour justifier cette formule, on peut montrer par récurrence que pour tout n il existe un flottant x tel que $d_n \leq e_n + \epsilon^2 x$.

Théorème 8.8 *La séquence de flottants $x_1, ..., x_n$ minimise e_n lorsque les x_i sont triés par ordre décroissant.*

Un pré-traitement qui trie en $\mathcal{O}(n \log n)$ les n nombres flottants permet alors de calculer la somme en minimisant l'erreur e_n.

C Balayage de Graham

Étant donné un ensemble de points sur le plan, on cherche une enveloppe convexe de ces points. Une solution, appelée **balayage de Graham**, cherche le point p_0 d'ordonnée et d'abscisse minimales, puis trie les sommets par angle polaire respectivement à p_0. Ce pré-traitement nous permet de calculer l'enveloppe convexe de proche en proche.

Leçon 9

Algorithmique du texte. Exemples et applications.

Auteur·e·s: Sorci Émile
Niveau : L2 niveau MPI
Pré-requis : Structures arborescentes, Automates finis, Notions d'algorithmique (Programmation dynamique et gloutons)
Références : [Cormen et al., 2009], [Beauquier et al., 2005],[Crochemore and Rytter, 1994], [Gusfield et al., 1997],[Dumas et al., 2007a]

I Introduction

A Motivation

L'algorithmique de texte trouve de nombreux champs d'applications, comme la bio-informatique (séquençage d'ADN, biologie moléculaire), la recherche dans une base de donnée ou de motifs dans un texte (commande grep et C-f), l'analyse lexicale en compilation, le traitement du langage naturel (autocomplétion, correction automatique des erreurs etc...) ou encore la compression.

Ce domaine utilise et applique une multitude de concepts informatiques et algorithmiques : glouton, programmation dynamique, automates, structures de données, arbres.

B Définitions de base

Définition 9.1

Un **alphabet** Σ *est un ensemble fini. Un* **mot** *sur* Σ *est une séquence finie de caractères de* Σ. Σ^* *est l'***ensemble des mots** *sur* Σ. *Le* **mot vide** *est noté* ϵ.

- y *est préfixe d'un mot* x *s'il existe* z *tel que* $x = yz$.
- z *est un suffixe d'un mot* x *s'il existe* y *tel que* $x = yz$
- w *est un facteur d'un mot* x *s'il existe* y *et* z *tels que* $x = ywz$.

Si x *est de taille* n, *pour tous* $0 < i < j \leq n$ *on notera* $x[i...j] = x_i x_{i+1} \dots x_j$. *Si* $j < i$, *on a* $x[i..j] = \epsilon$.

On appellera parfois **texte** est un mot de grande taille.

II Représentation des textes et ensembles de chaînes de caractères

Chaque représentation a plusieurs applications qui la rendent utile.

A Arbre des préfixes ou trie

Définition 9.2 (Arbre des préfixes) *Etant donné un ensemble de mots X, l'***arbre des préfixe** *P de cet ensemble est défini comme suit :*

- *L'ensemble des sommets de P est l'ensemble des préfixes des mots de X ;*
- *Il existe une arête du préfixe de p à p' dans P si $p' = p \cdot a$ avec $a \in \Sigma$.*

Proposition 9.1 *Un arbre préfixe est une implémentation d'un ensemble. Pour un mot de taille n, la recherche, l'insertion et la suppression se font en $O(n)$. La création est en $O(\Sigma_{x \in X}|x|)$.*

Exemple 9.1 *On crée l'arbre des préfixes associé à l'ensemble $\{tarte, tartine, pates, patates, tarot, tri\}$.*

Application 9.1 *En traitement du langage naturel : autocomplétion, détection de lexiques dans un texte ; Algorithmique : trier rapidement un ensemble de chaines de caractères.*

B Arbres suffixes

Définition 9.3 (Arbre des suffixes) *Un* **arbre des suffixe** *d'un mot x est l'arbre préfixe de l'ensemble des suffixes de x.*

Exemple 9.2 *On donne l'arbre des suffixes du mot abccaba.*

Définition 9.4 (Arbre des suffixes généralisés) *Un* **arbre suffixe généralisé (ASG)** *d'un ensemble de mots X est l'arbre préfixe de l'ensemble des suffixes des mots de X.*

Proposition 9.2 *Un ASG d'un ensemble X se construit naïvement en $O(\Sigma_{x \in X}|x|^2)$ dans le pire cas.*

Remarque 9.1 *Il existe des algorithmes de construction en temps linéaire.*

Application 9.2
- *Trouver les occurences d'un ou plusieurs motifs dans un ensemble de mots ;*
- *Trouver la plus longue sous chaîne commune à un ensemble de mots en temps linéaire en la somme des mots, ce qui améliore une approche naïve par programmation dynamique.*

C Compression

On dispose d'un texte t et on souhaite lui donner une représentation en mémoire de taille faible.

Généralités sur les codes

Définition 9.5 (Codes) *Un* **codage** *pour un alphabet Σ est une fonction injective $h : \Sigma \to \{0,1\}^*$. Pour un caractère c, $h(c)$ est le* **mot de code** *de c. On étend h à Σ^+ en posant $h(x) = h(x_1) \dots h(x_m)$ pour tout mot $x \in \Sigma^+$ de taille m.*

Un codage est dit :

— *de* **longueur fixe** *si tous les mots de code sont de même longueur ;*
— *de* **longueur variable** *sinon ;*
— *un code est* **non-ambigu** *si tout mot de* $\{0,1\}*$ *peut être décodé d'au plus une seule façon ;*
— **préfixe** *si aucun codage n'est préfixe d'un autre.*

Remarque 9.2 *Un codage préfixe est non ambigu.*

Exemple 9.3 *Exemple de codage à longueur variable et préfixe pour* $\Sigma = \{a, b, c, d, e, f\}$ *est la fonction* χ *définie par* $\chi(a) = 0$, $\chi(b) = 101$, $\chi(c) = 100$, $\chi(d) = 111$, $\chi(e) = 1101$ *et* $\chi(f) = 1100$.

Codage de Huffman

Idée 9.1 Pour le codage *: étant donné un texte* t*, on souhaite le représenter avec le moins de bits possibles à l'aide d'un code préfixe de longueur variable.*
Pour l'algorithme *: construire de façon gloutonne un arbre dont les feuilles sont les éléments de* Σ *et dont le chemin de la racine à une feuille donne le mot de code du caractère (enfant gauche* $\rightarrow$ *0 et enfant droit* $\rightarrow$ *1).*

Algorithme de Huffman

Algorithme 9.1 : Algorithme de Huffman

Données : Un texte t.
Résultat : Un arbre qui représente le code de Huffman de t.
$T = \{(c, freq(c,t)), \quad c \in \Sigma\}$;
$\#$ T *contient un ensemble d'arbres que nous allons fusionner itérativement*
tant que $|T| > 1$ **faire**
  choisir $t_1, t_2 \in T$ de fréquence minimale ;
  Remplacer t_1 et t_2 dans T par un éléménent t_3 de fréquence $f(t3) = f(t_1) + f(t_2)$;
retourner *T[0]*

Proposition 9.3 *L'algorithme de Huffman produit un codage préfixe de longueur variable en temps* $\mathcal{O}(|\Sigma| \log(|\Sigma|) + t)$ *en utilisant un tas.*

Théorème 9.1 *Soit* h *le code de Huffman d'un texte t et* l *un autre codage des caractères non ambigü. Alors,* $|h(t)| \leq |l(t)|$.

Remarque 9.3 *En particulier, cela signifie que* h *effectue bien une compression efficace.*

Proposition 9.4 *L'arbre permet d'encoder et décoder un texte en temps linéaire.*

Remarque 9.4 *En pratique, il faut stocker l'arbre avec le texte compressé.*

Application 9.3 *Est utilisé à certaines étapes de la compression dans les formats JPEG, MP3 et zip par exemple.*

Algorithme de Lempel, Ziv, Welch On part d'un dictionnaire qui à chaque caractère associe un entier encodé sur un nombre fixe de bits (ex : ASCII) et on l'étend dynamiquement aux chaînes de caractères lors de la lecture de texte.

1. on lit les symboles jusqu'à former un mot qui n'est pas dans le dictionnaire ;
2. on l'ajoute au dictionnaire avec le prochain code libre (257 en premier) ;
3. on écrit le code correspondant au mot sans le dernier caractère ;
4. on recommence en partant du dernier caractère.

Exemple 9.4 *Faire trouver l'algorithme de compression et décompression sur un texte exemple : taratata.*

Proposition 9.5 *On construit le dictionnaire dynamiquement de telle sorte qu'il n'est pas nécessaire au décodage. Il est même reconstruit dynamiquement au décodage.*

Présentation des algorithmes d'encodage et décodage :

Exemple 9.5 *Pour un texte de la forme $(abcd)^k$, l'algorithme de Huffman produit un code de taille $8k$ (2 bits par caractère), si l'on prend un dictionnaire sur 3 bits pour LZW, pour k assez grand, le code produit par LZW utilise environs $\frac{3}{2}$ bits par caractère.*

III Problèmes de similarité des séquences

A Plus longue sous suite commune (PLSC)

Définition 9.6 *Un mot s de taille k est une* **sous-séquence** *de x de taille m s'il existe une famille strictement croissante d'indices $\{i_1, \cdots, i_k\}$ telle que pour tout $j, x_{i_j} = s_j$.*

Exemple 9.6 *ada et adcb sont des sous-suites de abdcab.*

La problème : étant donnés x et y deux mots sur un alphabet Σ, donner une PLSC de x et y.
Résolution du problème : on utilise un algorithme de programmation dynamique.

B Distance d'édition

Définition 9.7 *On définit les opérations de substitution, insertion et suppression dans une chaîne de caractères comme* **opération d'édition**. *La* **distance de Levenshtein** *entre deux mots est le nombre minimal d'opérations d'édition nécessaires pour passer de l'une à l'autre.*

Développement 7. Calcul de la distance d'édition.

Remarque 9.5 *Dans la section suivante, on aborde la recherche de motif. On cherche ainsi sous-chaîne de notre texte à distance nulle de notre motif. Si on allège cette contrainte (en bornant la distance par exemple), on peut faire de la recherche de motif « approchée ».*

IV Recherche d'un motif dans un texte

A Définition du problème et notion de fenêtre

Étant donné un texte T et un motif M, le but est ici de déterminer si M est un facteur de T, autrement dit si M apparaît dans T. Si besoin, on pourra aussi calculer le nombre d'occurrences de M dans T.

Dans toute la partie on aura $|M| = k$ et $|T| = n$.

Exemple 9.7 *Avec $T = aababababbbbabababab$ et $M = abab$, les occurrences sont $\{2, 4, 6, 8, 13, 15\}$.*

Dans les algorithmes que l'on va étudier, on observera une fenêtre de taille m du texte.

B Algorithme naïf

Pour tout caractère de T, on teste s'il correspond à un occurrence de M.

Proposition 9.6 *Dans le pire cas, la complexité temporelle est $\mathcal{O}(mn)$. Pour un texte aléatoire, l'algorithme est en $\mathcal{O}(2n)$ en moyenne.*

Remarque 9.6 *Même si cette complexité linéaire peut sembler satisfaisante, les textes sont souvent de très grande taille. On va donc essayer d'améliorer la complexité moyenne de cet algorithme.*

C Algorithme de Boyer-Moore, version de Horspool

L'idée est de lire de gauche à droite le texte T et M de droite à gauche ce qui nous permet de sauter des parties entières des mots. Trois cas peuvent se produire :

— tous les caractères sont égaux, on a trouvé une occurrence et on décale la fenêtre au maximum ;

— si on trouve une différence à la lettre t_i et que $t_i \notin M$, on décale la fenêtre à l'indice $i + 1$.

— si on trouve un différence à la lettre t_i avec $t_i \in M$, on décale la fenêtre pour aligner la première occurrence de t_i dans M à l'indice i.

Algorithme 9.2 : Algorithme de Horspool

Données : Un texte t et un motif x.
Résultat : L'ensemble des occurences de x dans t.
pour j *allant de* 1 *à* $n - m + 1$ **faire**
  si $t[j..j + m]! = x$ **alors**
    Décaler x pour aligner $t[j + m]$ et l'occurence de $t[j + m]$ la plus à droite dans x ;
  sinon
    Ajouter j aux occurences de x dans t et décaler x à la deuxième occurence la plus à droite de $t[j + m]$ dans x ;

Pour cet algorithme, il faut pré-calculer l'ensemble des décalages.

Proposition 9.7 *Dans le pire cas, la complexité est en $\mathcal{O}(mn)$.*

Remarque 9.7 *En pratique, l'algorithme est très rapide (sous linéaire en moyenne) et très utilisé.*

Exemple 9.8 *Pire cas en $T = a^n$ et $M = ba^{m-1}$ et cas rapide avec $T = aabbbababacaabbab$ et $M = aababab$.*

D Automate des motifs

On peut construire un automate qui reconnaît le langage le langage $\Sigma^* M$. Ainsi, on simule A sur l'entrée T.

Développement 8. Construction de l'automate des motifs.

E Généralisations du problème

Recherche d'un ensemble fini de motifs dans un texte. Soit X un ensemble de motifs et t un texte. L'algorithme d'**Aho-Corasick** qui reprend l'idée de trie et d'automate des occurrences permet de résoudre ce problème en temps $\mathcal{O}(m+n)$ où $m = \Sigma_{x \in X}|x|$ et $\mathcal{O}(m)$ en espace.

Appartenance d'un mot au langage d'une expression régulière. Le problème de recherche d'un motif x dans un texte t revient à décider si $t \in \Sigma^* x \Sigma^*$. On peut généraliser au problème suivant : étant donné une expression régulière e et un mot t, est-ce-que $t \in L(e)$? Pour cela, on crée un automate fini déterministe à partir de l'expression régulière.

Application 9.4 *Ce problème est le problème de base de l'analyse lexicale, qui est centrale en compilation.*

Appartenance d'un mot au langage d'une grammaire. De la même manière, on peut se demander si un mot peut être engendré par une grammaire. L'algorithme **Cocke-Younger-Kasami** (CYK) permet de résoudre ce problème dans le cadre des grammaires sous forme normale de Chomsky.

Application 9.5 *Ce problème est le problème de base de l'analyse syntaxique, qui est centrale en compilation.*

Leçon 10

Arbres : représentations et applications.

Auteur·e·s: Bertrand Jules, Marin Malory, Sorci Émile
Niveau : L1
Pré-requis : Structures de données basiques
Références : [Gaudel et al., 1990],[Goodrich and Tamassia, 2014]

I Motivation et définitions

De manière abstraite, un **arbre** est une structure de donnée qui stocke des éléments de manière **hiérarchique**. À part l'élément du haut, chaque élément a un **parent** et possiblement des **enfants**. L'élément du haut est appelé **racine** et on représente un arbre du haut vers le bas (contrairement aux arbres en biologie).

Exemple 10.1 (Arborescence des fichiers) *Un système de fichier peut être représenté par un arbre, où les enfants d'un fichiers est l'ensemble des fichiers qu'il contient.*

Exemple 10.2 (Expressions arithmétiques) *Une expression arithmétique, comme par exemple* $(2+3)\times(5+7)$*, se représente naturellement par un arbre. Les parenthèses servent juste à la hiérarchie.*

A Définitions

Définition 10.1 (Arbre binaire) *Un* **arbre binaire** *est soit vide (noté* `Nil`*), soit de la forme* $B = (u, B_1, B_2)$*, où* B_1 *et* B_2 *sont des arbres binaires disjoints et* u *est un nœud appelé* **racine** *de* B*.*

Remarque 10.1

1. *Cette définition n'est pas symétrique, l'arbre* $(u, (u_1, \mathit{Nil}, \mathit{Nil}), (u_2, \mathit{Nil}, \mathit{Nil}))$ *n'est pas la même chose que* $(u, (u_2, \mathit{Nil}, \mathit{Nil}), (u_1, \mathit{Nil}, \mathit{Nil}))$*.*
2. B_1 *(resp.* B_2*) est appelé* **sous-arbre gauche** *(resp.* **droit***) de* $B = (u, B_1, B_2)$*. La racine de* B_1 *(resp.* B_2*) est appelé* **fils gauche** *(resp.* **droite***).*
3. *Un nœud racine d'un arbre de la forme* $(u, \mathit{Nil}, \mathit{Nil})$ *et appelé* **feuille***.*

On peut généraliser ces définitions.

Définition 10.2 (Arbre) *Un* **arbre** *est soit l'arbre vide* `Nil`*, soit* $A = (u, A_1, ..., A_p)$ *où* $p \geq 1$*,* $A_1, ..., A_p$ *sont des arbres et* u *est un nœud.*

Si à chaque nœud d'un arbre on associe une étiquette (un entier par exemple), on parle d'**arbre étiqueté**.

Exercice 10.1 *Définir un type* `binaryTree` *en OCaml.*

B Mesures sur les arbres

On présente quelques mesures classiques sur les arbres binaires. Les définitions s'étendent facilement aux arbres quelconques.

Définition 10.3 (Taille) *La* **taille** $t(A)$ *d'un arbre binaire* A *est le nombre de nœud qui le compose. On peut la définir inductivement :*

— $t(\mathit{Nil}) = 0$,

— $t(u, A_1, A_2) = 1 + t(A_1) + t(A_2)$.

Définition 10.4 (Hauteur) *La* **hauteur** $h(A)$ *d'un arbre* A *est la taille d'un plus long chemin de la racine jusqu'à une feuille. On peut la définir de manière inductive :*

— $h(\mathit{Nil}) = 0$,

— $h(u, A_1, A_2) = 1 + \max(h(A_1, A_2)$.

Exercice 10.2 *Écrire deux fonctions* `taille : binaryTree -> int` *et* `hauteur : binaryTree -> int`.

Exercice 10.3 (Lemme de König) *Tout arbre infini à branchement fini contient un chemin infini.*

Lemme 10.1 *Pour un arbre binaire ayant* n *nœuds au total et de hauteur* h*, on a :*

$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Proposition 10.1 *Le nombre d'arbres binaires de taille* n *est* $b_n = \frac{1}{n+1}\binom{2n}{n}$.

C Arbres binaires particuliers

On appelle **arbre complet** un arbre pour lequel chaque niveau est complètement rempli. Un **arbre parfait** (ou **complet gauche**) possède chaque niveau rempli sauf éventuellement le dernier, qui est lui partiellement rempli à gauche. Enfin, un **peigne** gauche est un arbre dans lequel chaque nœud ne possède pas de fils droit.

Exercice 10.4 (Arbres filiformes) *Un arbre filiforme est une généralisation des peignes a au moins un enfant qui est une feuille. Combien existe-t-il d'arbres filiformes à* n *nœuds ?*

II Représentation et parcours

A Représentation des arbres binaires

Utilisation des pointeurs. À chaque nœud, on associe deux pointeurs, l'un vers le sous-arbre gauche, l'un vers le sous-arbre droit. L'arbre est déterminé par l'adresse de sa racine.

Utilisation de tableaux. Lorsqu'on ne peut pas allouer dynamiquement de la mémoire, on peut implémenter l'arbre binaire via un tableau à trois colonnes (noeud, fils droit et fils gauche). Cette structure permet notamment de stocker un arbre dans une base de donnée relationnelle.

Représentation des arbres binaires parfaits. Si un tel arbre contient n nœuds, il peut être représenté par un tableau de taille n indicé de 1 à n. Ainsi, si un nœud est à la case $1 \leq i \leq n$, ses enfants seront aux cases $2i$ et $2i+1$.

B Représentation des arbres généraux

Globalement, représenter des arbres généraux n'est pas évident. L'une des seuls solutions valables consiste à généraliser la méthode des pointeurs. Lé théorème ci-dessous affirme qu'il existe d'une bijection entre les arbres binaires et les arbres généraux. En pratique, on peut donc représenter tout arbre par un arbre binaire.

Théorème 10.1 *Il existe une bijection entre les arbres généraux ayant $n+1$ nœuds et les arbres binaires à n nœuds.*

C Parcours d'arbre binaire

Parmi les opérations les plus mises en œuvre par des algorithmes sur les arbres figurent les parcours. Un parcours consiste à examiner systématiquement l'ensemble des nœuds de l'arbre, comme par exemple pour afficher le contenu de chaque nœud.

Parcours en profondeur. L'idée est d'aller le plus loin possible et de revenir en arrière lorsqu'on rencontre une feuille.

Algorithme 10.1 : ParcoursProfondeur(A)

```
si A est vide alors
    TraitementFeuille() ;
sinon
    Traitement1() ;
    ParcoursProfondeur(A.droit) ;
    Traitement2() ;
    ParcoursProfondeur(A.gauche) ;
    Traitement3() ;
```

Il y a trois ordres de parcours classique classiques :

— ordre préfixe : Traitement2() et Traitement3() n'existent pas ;

— ordre infixe : Traitement1() et Traitement3() n'existent pas ;

— ordre postfixe : Traitement1() et Traitement2() n'existent pas.

Exemple 10.3 (Parcours d'une expression arithmétique) *On retrouve les représentations habituelles : infixe (avec nécessité de parenthèse pour les priorités), et polonaises préfixée ou postfixée. La notation polonaise enlève toute ambiguïté.*

Parcours en largeur. Un parcours en largeur correspond à un ordre hiérarchique de l'arbre. Cet algorithme est naturellement itératif et peut s'implanter via une file.

III Arbre de recherche

L'une des principale utilisation des arbres est la notion de structure de donnée. En particulier (un peu comme la recherche dichotomique), on peut facilement implanter un dictionnaire avec des arbres.

A Arbre binaire de recherche

L'idée générale est de stocker les couples (k, x) dans un arbre, de telle sorte qu'il soit facile de retrouver un élément en partant de la racine.

Définition 10.5 (Arbre binaire de recherche) *Soit A un arbre binaire enraciné, étiqueté par une fonction $e : A \to U$. On dit que A est un* **arbre binaire de recherche** *(ou* **ABR***) si pour tout nœud n, en notant g et d ses fils gauche et droit (s'ils existent), on a $e(g) < e(n) < e(d)$.*

De tels ABR ne peuvent que stocker des clés, mais on modifie facilement la structure pour pouvoir rajouter des données satellites sur les nœuds, et donc implémenter les dictionnaires. Cette structure nécessite que l'ensemble où les clés vivent soit muni d'un ordre total.

Recherche et insertion. Dans un ABR, on insère un élément x en descendant depuis la racine, en prenant le fils gauche ou le fils droit selon si x est plus petit ou plus grand que la racine courante, et en créant un nouveau nœud étiqueté par x lorsque l'on ne peut plus avancer. L'opération de recherche se fait de manière analogue.

Exercice 10.5 *Définir un type* `tree` *en OCaml où chaque nœud contient un entier. Écrire les fonctions* `recherche : tree -> bool` *et* `insert : tree -> int -> tree`.

Suppression d'un élément. La suppression est un peu plus subtile. Lorsque le nœud est une feuille ou si le nœud n'a qu'un enfant, alors la transformation est simple. Par contre, si le nœud possède deux enfants, alors il faut retirer le minimum du sous-arbre droit (ou le maximum du sous-arbre gauche) pour le remplacer.

Exercice 10.6 *Écrire une fonction* `extraire_min : tree -> (tree*int)` *qui, étant donné un ABR, retourne le couple correspondant au minimum et à l'arbre obtenu en le supprimant. En déduire une fonction* `supprime : tree -> int -> tree`.

— Arbre parfaitement équilibré : soit A un ABR contenant n valeurs, tel que chaque feuille est à la même hauteur h. Alors, $h = O(\log(n))$, et donc l'insertion, la recherche se font en $(O(\log(n))$.

— Arbre linéaire : Soit A un ABR contenant n valeurs, tel qu'aucun nœud n'a de fils gauche. Alors, sa hauteur est $h = n$: L'insertion et la recherche se font en $O(n)$.

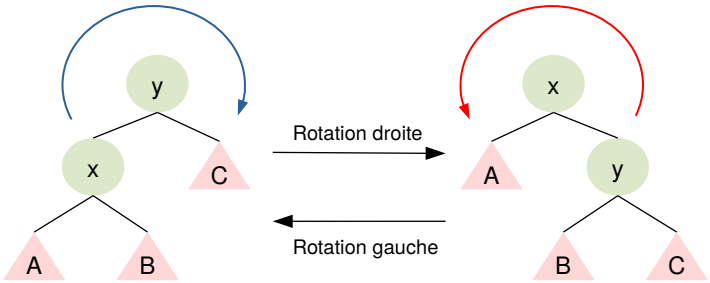
Complexité. Cet exemple souligne un problème majeur de l'implémentation par ABR : l'ordre d'insertion des éléments détermine le squelette de l'arbre, et on peut obtenir un peigne.

Proposition 10.2 *Soit A un ABR à n nœud et de hauteur h. La recherche, l'insertion et la suppression se font dans le pire cas en $\mathcal{O}(h)$ comparaisons.*

B Arbres équilibrés.

Arbres AVL. Une solution est de rééquilibrer l'arbre de recherche au fil des insertions, et de garantir à chaque instant une hauteur $h = O(\log(n))$. Une telle solution a été notamment implémenté dans les **AVL** (Adelson-Velsky et Landis) dont les mécanismes de rééquilibrages se basent sur des opérations de rotation d'arbres. Un tel arbre est dit automatiquement équilibré.

Exemple 10.4



B-arbres. Dans certains cas, le côté « binaire » des arbres de recherches peuvent poser problème. En effet, regardons le cas d'une hiérarchie mémoire où les données sont stockés dans une mémoire lente (comme le disque par exemple). Au lieu de rapatrier un nœud après l'autre, il peut être plus intéressant de créer de plus « gros nœuds ».
Développement 3 : B-arbre

IV Les arbres en théorie des graphes

Définition 10.6 (Arbre) *On appelle* **arbre** *un graphe non orienté, connexe et acyclique.*

Proposition 10.3 *Soit $G = (V, E)$ un graphe non orienté, ayant n sommets. Les propriétés suivantes sont équivalentes :*

1. *G est un graphe connexe sans cycle,*
2. *G est connexe avec $(n-1)$ arêtes,*
3. *G est acyclique avec $(n-1)$ arêtes.*

Étant donné un graphe $G = (V, E)$ connexe, on appelle **arbre couvrant** de G tout sous-graphe connexe et acyclique de G. Lorsque le graphe est pondéré, un problème classique est la **recherche d'un arbre couvrant de poids minimum**. On étudie deux algorithmes permettant de résoudre ce problème : celui de Kruskal et celui de Prim.

A Algorithme de Prim et file de priorité [Cormen et al., 2009]

Algorithme de Prim

Algorithme 10.2 : Algorithme de Prim

Données : Un graphe pondéré G.
Résultat : Un arbre couvrant de poids minimal T.
Initialiser T un arbre vide enraciné au somment 0 de G ;
tant que *il existe des sommets de G qui n'ont pas été mis dans l'arbre* **faire**
  Choisir t dans $G \backslash T$, accessible depuis T et qui minimise la distance avec les éléments de T ;
  Placer t dans T ;
retourner T

L'algorithme de Prim sélectionne itérativement et de façon gloutonne les sommets les plus proches d'un sous arbre T de G. Il utilise un front de sommets éligibles au cours de son exécution.

Théorème 10.2 *L'algorithme de Prim est correct.*

Besoin de deux structures de données : une pour le graphe et une seconde que l'on appelle file de priorité.

La structure de donnée.

Définition 10.7 *Une* **file de priorité** *est une structure qui permet d'organiser un ensemble d'éléments munis de clés, l'ensemble de clés étant ordonné. Une file de priorité* `Q` *est définie pour tout élément* `x` *associé à une clé* `k` *par les opérations* `Inserer(Q, x, k)`, `Extraire_min(Q)`, `Est_vide(Q)` *et* `Réduire_clé(Q, x, k)`.

Une première implémentation des files de priorités utilise simplement un tableau. On présente une manière plus efficace : un tas.

Implémentation de la structure de file de priorité avec un tas

Définition 10.8 *Un* **tas-min** *(resp. max) est un arbre binaire complet gauche vérifiant la propriété du tas-min (resp. max) : la clé d'un nœud est inférieur à la clé de ses enfants.*

Exemple 10.5 *On donne un tas-max pour la liste [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]. On donne un tas-min avec ces éléments ainsi que plusieurs exemples d'arbres qui ne sont pas des tas.*

Comme on l'a vu précédemment, un arbre binaire complet gauche peut facilement être représenté via un tableau. Si le tableau est trié, alors le tas correspondant vérifie la propriété du tas-min.

On fournit l'implémentation de `Inserer(Q, x, k)`, `Extraire_min(Q)` et `Réduire_clé(Q, x, k)`. On montre qu'elles se font en $O(log(n))$.

Complexité de Prim.

Proposition 10.4 *Avec une implémentation de tas-min pour la file de priorité et de liste d'adjacence pour le graphe, l'algorithme de Prim est en* $O(|E| + |V| \log(|V|))$.

Développement 4. Autre application du tas : création d'un tas en $O(n)$ et tri par tas

Application 10.1 *Deux autres applications des files de priorité :*

- *L'algorithme de Dijkstra ;*
- *L'ordonnancement des taches avec priorité (attention, c'est pas toujours des tas, par exemple l'ordonnanceur linux utilise des arbres rouge noir)*

B Algorithme de Kruskal et structure Unir et Trouver

Algorithme de Kruskal

Algorithme 10.3 : Algorithme de Kruskal

Données : Un graphe pondéré G.
Résultat : Un arbre couvrant de poids minimal T.
Initialiser T un forêt sans arcs avec les sommets de G ;
tant que *il existe des sommets de* G *qui n'ont pas été mis dans l'arbre* **faire**
  Choisir e une arête de G de poids minimal qui ne crée pas de cycle dans T ;
  Ajouter e à T ;
retourner T

L'algorithme de Kruskal sélectionne itérativement, de façon gloutonne les arêtes avec les poids les plus faibles et qui ne créent pas de cycle. On construit une forêt qui petit à petit se réunit pour former un seul arbre à la fin de l'exécution de l'algorithme.

Théorème 10.3 *L'algorithme de Kruskal est correct.*

Remarque 10.2 *On ré-utilise une structure que l'on a déjà vu dans le passé, la file de priorité pour stocker les arêtes dont les étiquettes sont leur poids.*

Il nous faut une manière efficace de déterminer si l'ajout d'une arête à la forêt T crée un cycle ou non. Pour cela nous allons introduire une nouvelle structure de données : unir et trouver.

Définition de la structure unir et trouver [Dasgupta et al., 2008]
La structure unir et trouver permet de représenter et travailler avec des partitions d'un ensemble.

Définition 10.9 *La structure unir et trouvée structure un ensemble E de n éléments arrangés en partitions $P_1, \cdots, P_i$ de $E = \sqcup_{i=1}^{n} P_i$ où chaque partie P_i a un représentant r_i. Elle est définie par les opérations* `Unir(E, i, j)` *qui fait l'union ds parties P_i et P_j et choisit un représentant parmi cette nouvelle partie et* `Trouver(E, x)` *qui renvoie le représentant de x dans E. On ajoute aussi l'opération* `Créer_ensemble(E, x)` *qui ajoute l'ensemble $\{x\}$ à la structure si x n'y appartient pas déjà et crée un ensemble singleton dont le représentant est x.*

Exemple 10.6 *En partant d'un ensemble simple $\{3, 9, 18, 33, 208\}$ on crée la structure d'unir et trouver et on effectue quelques opérations unir et trouver.*
On donne l'exemple de l'algorithme de détection de composantes connexes dans un graphe.

Remarque 10.3 *Même dans des textes écrits en français, il ne sera pas rare de lire Union-Find plutôt que unir et trouver. En anglais on parle aussi de structure d'ensemble disjoints.*

Théorème 10.4 (Complexité) *L'algorithme de Kruskal utilise $|V|$ fois* `CréerEnsemble`*, $2|E|$ fois* `Trouver` *et $|V| - 1$ fois* `Union`*.*

Implémentation avec des forêts Un ensemble est représenté par un arbre, chaque nœud est un élément et il pointe vers son parent. La racine de l'arbre est le représentant de l'ensemble.

Avec cette représentation, on peut facilement faire `CreerEnsemble`. En revanche, l'implémentation de `unir` et `Trouver` n'est pas efficace, puisqu'il faut remonter l'arbre.

Optimisation 1 : Union par rang. On associe à chaque racine un rang, correspondant à la hauteur de son arbre. En s'assurant que lors d'une union, on augmente au maximum la hauteur de 1, on peut avoir une borne sur la hauteur maximal d'un arbre et donc les opérations `Unir` et `Trouver` en $\mathcal{O}(\log n)$.

Théorème 10.5 *Avec une implémentation de la structure Unir et trouver par une forêt avec union par rang, l'algorithme de Kruskal s'exécute en temps $\mathcal{O}(|V| + |E|) \log n)$.*

Remarque 10.4 *On atteint le temps de tri des arêtes qui est incompressible. Par contre, si on suppose que les arêtes sont déjà triées, on peut encore améliorer la complexité.*

Optimisation 2 : Compression de chemin. À chaque `Trouver`, on peut compresser l'arbre. On obtient une bien meilleure complexité amortie.

Application 10.2 — *On l'applique à l'algorithme de Kruskal ;*

— *On l'applique au problème de connectivité dynamique ;*

— *On l'applique à l'algorithme de minimisation d'un automate.*

Leçon 11

Exemples d'algorithmes d'approximation et d'algorithmes probabilistes.

Auteur·e·s: Marin Malory
Niveau : L3
Pré-requis : algorithmique, NP-complétude, probabilités
Références : [Cormen et al., 2009], [Mitzenmacher and Upfal, 2005], [Benoit et al., 2013], [Motwani and Raghavan, 1995]

Dans le cadre de cette leçon, on s'intéresse à la recherche d'une solution à un problème en relâchant certaines contraintes. Dans la première partie, on autorise notre algorithme à avoir une partie aléatoire, pouvant causer un temps d'exécution ou un résultat aléatoire. Dans un second temps, on s'autorisera à renvoyer un résultat différent du résultat optimal, tant qu'il reste assez proche.

I Algorithmes probabilistes

A Algorithmes déterministes/probabilistes

Définition 11.1 (Algorithme déterministe) *Un algorithme déterministe est un algorithme qui, étant donné une entrée x, produit toujours la même exécution.*

Définition 11.2 (Algorithme probabiliste) *Un algorithme probabiliste est un algorithme qui, étant donné une entrée x, produit une sortie qui dépend non seulement de x mais aussi de valeur obtenue via un* **générateur de nombre aléatoire**.

Remarque 11.1 *En pratique, la plupart des environnements de programmation propose un* **générateur de nombres pseudo-aléatoires**, *qui est un algorithme déterministe renvoyant des nombres qui ont l'air aléatoire.*

Remarque 11.2 *Le sens de déterministe ci-dessus n'est pas le même que celui utilisé pour les machines de Turing. En effet, une machine de Turing non-déterministe renvoie toujours la même valeur sur une même entrée.*

B Exemples

On commence ici par donner deux exemples : le tri rapide randomisé, et un algorithme probabiliste pour trouver une coupe minimale.

Tri rapide randomisé Dans cette version du tri rapide, on choisit le pivot de manière aléatoire.

Algorithme 11.1 : TriRapideRandomisé(T)

```
si |T| = 0 ou |T| = 1 alors
    retourner T
sinon
    # On choisit un pivot aléatoirement
    pivot ← Random({0, ..., |T| − 1}) ;
    #On partitionne selon le pivot
    pivot ← Partition(T, pivot) ;
    T[: pivot[← TriRapideRandomisé(T[: pivot[) ;
    T[pivot + 1 :] ← TriRapideRandomisé(T[pivot + 1 :]) ;
    retourner T
```

Remarque 11.3 *Cet algorithme renvoie toujours un tableau trié, mais son temps d'exécution dépend des choix aléatoires du pivot. On parle d'algorithme du type* **Las Vegas**.

Théorème 11.1 *L'espérance du nombre de comparaisons du tri rapide randomisé d'un ensemble à n éléments est au plus $2nH_n$ où H_n est le terme général de la série harmonique.*

Coupe minimum randomisée

Définition 11.3 *Étant donné un graphe $G = (S, A)$ connexe, une coupe de G est un ensemble d'arêtes $A' \subset A$ tel que $G'(V, A')$ n'est pas connexe.*

L'algorithme randomisé pour obtenir une coupe minimum consiste à contracter des arêtes prises aléatoirement jusqu'à obtenir un graphe à deux sommets. On utilise pour cela une généralisation des graphes : les multigraphes. Son temps d'exécution est toujours un $\mathcal{O}(|S|)$ puisqu'on contracte exactement $|S| - 1$ arêtes, mais on ne retourne pas forcément une coupe minimum. On appelle cela un algorithme du type **Monte Carlo**.

Théorème 11.2 *L'algorithme randomisé décrit ci-dessus retourne un coupe minimale avec une probabilité $\geq \frac{2}{|S|^2}$.*

C Algorithme de type Monte Carlo et Las Vegas

Définition 11.4 (Algorithme Monte Carlo) *Un algorithme de Monte Carlo pour un problème $\mathcal{P}$ est un algorithme probabiliste A tel que pour toute instance $\mathcal{I}$ de $\mathcal{P}$:*

1. *$A(\mathcal{I})$ est une solution erronée avec une certaine probabilité,*
2. *le temps d'exécution de A sur $\mathcal{I}$ peut être borné indépendamment des choix aléatoires de A.*

Remarque 11.4 *Pour les problèmes de décision, il y a deux types d'algorithmes Monte Carlo : ceux à erreur unilatéral (one-sided error) et à erreur bilatéral (two-sided error).*

Exemple 11.1 (Hachage) *Algorithme Monte Carlo pour le problème de décision : existe-t-il $x \in X$ tel que $f(x) = y_0$, où X est un « grand » ensemble, $f : X \to \mathbb{R}$ une fonction de hachage et $y_0 \in \mathbb{R}$.*

Proposition 11.1 (Amplification) *Soient $0 < \epsilon_2 < \epsilon_1 < 1$. S'il existe un algorithme Monte Carlo pour un problème $\mathcal{P}$ ayant une probabilité d'erreur ϵ_1, alors on peut construire un algorithme Monte Carlo pour le problème $\mathcal{P}$ ayant une probabilité d'erreur ϵ_2.*

Définition 11.5 (Algorithme Las Vegas) *Un algorithme du type Las Vegas pour un problème $\mathcal{P}$ est un algorithme probabiliste A tel que pour toute instance $\mathcal{I}$ de $\mathcal{P}$:*

1. *$A(\mathcal{I})$ est une solution correcte à l'instance $\mathcal{I}$,*
2. *le temps d'exécution de A sur $\mathcal{I}$ est une variable aléatoire.*

Exemple 11.2 (Hachage) *Algorithme Las Vegas pour le problème de l'exemple précédent.*

Exercice 11.1 (De Monte Carlo à Las Vegas) *Étant donné un algorithme A du type Monte Carlo pour un problème $\mathcal{P}$ ayant une probabilité d'erreur ϵ, construire un algorithme Las Vegas pour $\mathcal{P}$. Que vaut l'espérance de son temps d'exécution ?*

II Algorithmes d'approximation

Lorsqu'on étudie un problème d'optimisation, il n'est pas toujours concevable de trouver une solution optimale en temps polynomiale. On utilisera alors des **algorithmes d'approximations** qui permettent de donner une solution « proche » de l'optimale tout en s'exécutant en temps polynomiale.

A Définitions

On s'intéresse ici exclusivement à des problèmes d'optimisation, dont on rappelle la définition.

Définition 11.6 (Problème d'optimisation) *Un problème d'optimisation est un problème $\mathcal{P} = (I, S)$ muni d'une* **fonction d'évaluation** *$c : I \times S \to \mathbb{R}^+$ calculable en temps polynomial et d'une* **fonction objectif** *$o \in \{\min, \max\}$.*

Étant donné une instance $i \in I$ de $\mathcal{P}$, l'objectif du problème d'optimisation $\mathcal{P}$ est de construire une solution $s^* \in S(i)$ vérifiant :

$$c(i, s^*) = o\{c(x, s) : s \in S(i)\}$$

Définition 11.7 (Algorithme d'approximation) *Une λ-approximation pour un problème d'optimisation $\mathcal{P}$ est un algorithme A ayant un temps d'exécution polynomial en la taille de l'instance et qui retourne une solution approximée qui est dans le pire des cas, à un facteur λ de la solution optimale. Autrement dit, pour toute instance $i \in I$, $A(i) \in S(i)$ et*

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \lambda$$

où $C = c(i, A(i))$ et $C^* = c(i, s^*)$ avec s^* une solution optimale pour l'instance i.

B Exemples

Couverture de sommets (Vertex cover).

Algorithme 11.2 : Glouton-VC

Données : Un graphe $G = (V, E)$.
Résultat : Un couverture des sommets de G, noté S.
$S \leftarrow \emptyset$;
tant que $\exists (u, v) \in E, u, v \notin S$ **faire**
  choisir $(u, v) \in E$ telle que $u, v \notin S$
  $S \leftarrow S \cup \{u, v\}$

Théorème 11.3 *L'algorithme Glouton-VC est une 2-approximation pour le problème de couverture de sommet.*

Définition 11.8 *Maximum Edge-Disjoint Paths (*$\mathbf{MEDP}$*)*
Entrée :

— *Graphe* $G = (V, E)$

— *Ensemble de requêtes* $\mathcal{R} \in \mathcal{P}(V \times V)$

Sortie :

$$\max_{A \subset \mathcal{R} \text{ est réalisable}} |A|$$

où $A \subset \mathcal{R}$ *est réalisable ssi pour tout* $r_i = (s_i, t_i) \in \mathcal{R}$ *il existe un chemin* π_i *de* s_i *à* t_i *dans* G*, et tel que pour tout* $(R_i, R_j) \in A^2$*, si* $i \neq j$ *alors* π_i *et* π_j *ne partagent aucune arête.*

On notera $A*$ *une solution optimale à ce problème.*

Théorème 11.4 (Admis) $\mathbf{MEDP}$ *est* $\mathbf{NP}$*-complet.*

Développement 9. Un algorithme d'approximation pour le problème $\mathbf{MEDP}$.

C Non-approximabilité

De la même manière que l'on peut montrer que des problèmes sont trop durs pour être résolus par des algorithmes en temps polynomial, on peut montrer que certains problèmes ne peuvent même pas être approximés. L'exemple le plus connu est le problème du **voyageur de commerce**.

Problème du voyageur de commerce.

Théorème 11.5 *Pour tout* $\lambda \geq 1$*, il n'existe aucune* λ*-approximation pour le problème du voyageur de commerce à moins que* $\mathbf{P} = \mathbf{NP}$.

Remarque 11.5 *La méthode générale pour prouver qu'il n'existe pas d'algorithme d'approximation est souvent le même que pour le théorème précédent : on suppose qu'un tel algorithme existe et on construit un algorithme polynomial qui résout un problème* $\mathbf{NP}$*-complet.*

Exercice 11.2 *Montrer que si on suppose que la fonction de coût satisfait l'inégalité triangulaire , on peut trouver une* 2*-approximation du problème du voyageur de commerce.*

D Algorithme d'approximation probabiliste pour satisfaisabilité MAX-3-CNF

Il est possible d'étendre la notion d'algorithme d'approximation aux algorithmes probabilistes en considérant C comme l'espérance du coût.

Définition 11.9 (MAX-3-CNF) *Étant donné n variables $x_1, x_2, ..., x_n$ et une formule sous forme normale conjonctive à m clauses contenant chacune 3 littéraux, maximiser le nombre de clauses satisfaite avec une assignation.*

Théorème 11.6 *Soit une instance de MAX-3-CNF à n variables $x_1, x_2, ..., x_n$ et m clauses ; alors, l'algorithme randomisé qui affecte indépendamment à chaque variable la valeur 1 avec une probabilité $1/2$ et la valeur 0 sinon est une $8/7$-approximation randomisée.*

Développement 10 : Introduction à la méthode probabiliste et lien avec les algorithmes d'approximations probabilistes.

Leçon 12

Stratégies algorithmiques (dont glouton, diviser pour régner, programmation dynamique, retour sur trace).

Auteur·e·s: Marin Malory, Sorci Émile
Niveau : L3
Pré-requis : niveau L2 informatique
Références : [Cormen et al., 2009], [Benoit et al., 2013]

Introduction

Face à un problème algorithmique, plusieurs choix de stratégies algorithmiques sont possibles. L'objectif de cette leçon est de donner un large éventail des différents paradigmes et leurs applications.

I Algorithmes gloutons

A Principe

Définition 12.1 *Un algorithme* **glouton** *est un algorithme qui, à chaque pas de calcul, fait un choix localement optimal.*

Exemple 12.1 *On considère un gymnase dans lequel on souhaite organiser n événements (s_i, e_i) (chaque événement est caractérisé par une date de début et une date de fin). Le but est de maximiser le nombre d'événements qui auront bien lieu dans le gymnase.*

- *choix 1 : l'événement qui minimise la durée $(e_i - s_i)$;*
- *choix 2 : l'événement qui a commence le plus tôt (qui minimise s_i) ;*
- *choix 3 : l'événement qui termine le plus tôt (qui minimise e_i).*

Chacun des choix mène à un algorithme glouton différent.

Exercice 12.1 *Montrer que les algorithmes gloutons faisant le choix 1 et 2 ne sont pas optimaux.*

B Algorithmes gloutons optimaux

Dans certains cas, ces choix localement optimaux mènent à un optimal global.

Exemple 12.2 *Dans le problème du gymnase, l'algorithme respectant toujours le choix 3 est optimal.*

Exercice 12.2 *Étant donné $A = (a_i)_{1 \leq i \leq n}$ et $B = (b_i)_{1 \leq i \leq n}$, donner un algorithme qui maximise $\prod_{i=1}^{n} a_i^{b_i}$.*

Arbres couvrants. Étant donné un graphe $G = (V, E, w)$, on cherche un arbre couvrant $T \subset E$ qui maximise $\sum_{e \in T} w(e)$.
Algorithme de Kruskal :

- on trie les arêtes par poids croissant ;
- pour chaque arête, la prendre si elle ne forme pas de cycle avec l'arbre en cours de construction.

Proposition 12.1 *L'algorithme de Kruskal est optimal.*

Remarque 12.1 *Cette proposition est directe si on montre que les forêts d'un graphe ont une structure de* **matroïde**.

C Algorithmes gloutons non-optimaux

Les algorithmes gloutons peuvent aussi donner de bons algorithmes d'approximations pour des problèmes $\mathbf{NP}$-complet.

Développement 20. Un algorithme glouton d'approximation pour un problème d'ordonnancement de tâches.

D Algorithme online

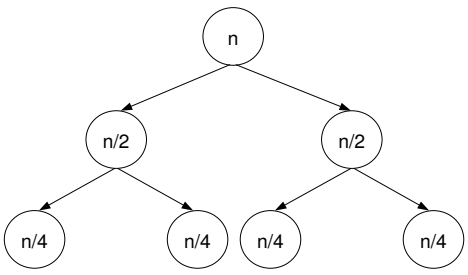
Certaines fois, la stratégie gloutonne est la seule possible puisque les données arrivent de manière indépendante. On peut donner plusieurs exemples :

— les algorithmes de remplacement de pages : FIFO ou LRU ;
— un algorithme d'ordonnancement qui exécute les tâches les plus tôt possible ;
— en apprentissage machine, l'algorithme 1-NN renvoie la valeur associée au plus proche voisin de la donnée reçue.

II Diviser pour régner

A Principe

Définition 12.2 *Un algorithme* **diviser pour régner** *résout le problème sur une instance de taille n en utilisant les résolutions sur des instances indépendantes plus petites ($n/2$ puis $n/4$ par exemple), et en fusionnant les résultats.*



Exemple 12.3 *Le tri-fusion est un algorithme diviser pour régner, permettant de trier n éléments en $\mathcal{O}(n \log n)$ comparaisons.*

Pour évaluer la complexité d'un tel algorithme, le théorème suivant est fondamental.

Théorème 12.1 (Master's theorem) *Soient $a, b \geq 1$, et T une suite vérifiant $T(n) = aT(n/b) + cn^\alpha$.*

- *Si $a > b^\alpha$, alors $T(n) = \Theta(n^{\log_b(a)})$;*
- *Si $a = b^\alpha$, alors $T(n) = \Theta(n^\alpha \log(n))$;*
- *Si $a < b^\alpha$, alors $T(n) = \Theta(n^\alpha)$.*

B Opérations arithmétiques efficaces

La stratégie « diviser pour régner » est très efficace pour les opérations arithmétiques, comme le produit de matrices par exemple.

Exemple 12.4

1. **Exponentiation rapide** :

$$x^n = \begin{cases} x^{n/2} \times x^{n/2} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x & \text{sinon} \end{cases}$$

2. *Algorithme de Strassen pour la multiplication de matrice.*

Remarque 12.2 *Certaines de ces opérations efficaces, comme l'addition binaire rapide, peuvent directement être implantées en machine.*

C Recherche efficace

La **dichotomie** est l'exemple le plus simple de la stratégie diviser pour régner. Les **arbres binaires de recherches** (ABR) utilisent ce principe au sein de leurs structures de donnée.

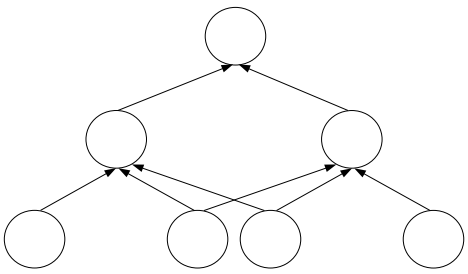
Exercice 12.3 *On considère une matrice M dans laquelle les coefficients sont triés par lignes et par colonnes. Donner un algorithme qui recherche un élément x dans M. Donner sa complexité.*

Exercice 12.4 *On considère n points $y_1, ..., y_n$. Donner un algorithme qui retourne $\text{argmax}_{1 \leq i < j \leq n}(y_j - y_i)$.*

III Programmation dynamique

A Principe

Définition 12.3 *Un algorithme de* **programmation dynamique** *résout des sous-instances du problème initial pour reconstruire une solution au problème global.*



Exemple 12.5 *Calcul de $\binom{n}{k}$.*

Remarque 12.3 *Pour éviter de calculer à nouveau les solutions intermédiaires, on peut les stocker. On parle de* **mémoïsation**.

B Application en théorie des graphes

On considère ici le problème des plus courts chemins dans un graphe orienté pondéré $G = (V, E, w)$ sans cycle de poids négatif.

L'**algorithme de Floyd-Warshall** résout ce problème via programmation dynamique. En posant, pour $(i, j) \in V^2$ et $0 \leq k \leq |V|$, $D_k[i, j]$ le poids du plus court chemin allant de i à j et passant seulement par les k premiers sommets, on peut trouver une relation de récurrence. L'algorithme obtenu réalise $\mathcal{O}(|V|^3)$ opérations élémentaires.

En général,

relation de récurrence $\rightsquigarrow$ algorithme de programmation dynamique

Exercice 12.5 *On définit la bande passante d'un chemin comme le minimum des poids des arêtes du ce chemin. Donner un algorithme qui, étant donné un graphe orienté pondéré, calcule la bande passante maximale entre deux sommets pour toute paire de sommets.*

C Application en algorithmique de texte

On considère ici deux problèmes que l'on retrouve en algorithmique de texte.

Commet mesurer la similarité entre deux chaînes de caractères ? Cette problématique trouve son sens en bio-informatique ainsi qu'en traitement de texte.

Plus longue sous-séquence commune.

Définition 12.4 *Étant donné $w = w_1...w_n$, une* **sous-séquence** *de w est un mot $w' = w_{i_1}...w_{i_k}$ avec $1 \leq i_1 < ... < i_k \leq n$.*

Définition 12.5 *Étant donné deux mots x et y, un mot z sous-séquence de x et de y de taille maximale est appelé* **plus longue sous-séquence commune** *de x et y.*

Proposition 12.2 *Étant donné deux mots x et y, on peut trouver une plus longue sous-séquence commune en $\mathcal{O}(|x|.|y|)$ opérations élémentaires.*

Distance d'édition. La distance d'édition entre deux mots mesure le nombre d'opérations élémentaires (insertion, suppression, substitution) permettant de passer d'une chaîne à une autre.
Développement 7. Calcul de la distance d'édition.

Comment savoir si un mot est engendré par une grammaire ? Cette problématique se pose en analyse syntaxique.

Définition 12.6 (Problème du mot) *Étant donné une grammaire $G = (\Sigma, V, S, R)$ et un mot $w \in \Sigma^*$, est-ce que $w \in L(G)$?*

L'algorithme de Cocke-Younger-Kasami (CYK) résout ce problème en temps cubique pour des grammaires sous forme normale de Chomsky. Pour cela, il faut poser $E_{ij} = \{A \in V | A \Rightarrow^* w_i ... w_j\}$.

IV Retour sur trace

A Principe

Définition 12.7 *Un algorithme de retour sur trace parcourt l'ensemble des solutions à un problème que l'on peut représenter sous la forme d'un arbre.*

L'algorithme peut ne pas parcourir tout l'arbre en s'arrêtant sur une branche si la solution en cours de construction n'est pas valide.

Application. Cette méthode est très utile lorsqu'on recherche une solution exacte à un problème tout en évitant une recherche par force brute.

Exemple 12.6 *Algorithme de résolution d'un sudoku par retour sur trace.*

B Application en théorie des graphes.

Définition 12.8 (Maximum Independent Set) *Étant donné un graphe $G = (V, E)$, un ensemble $V' \in V$ est dit* **indépendant** *si pour tout $i, j \in V'$, on a $(i, j) \notin E$. Le problème du Maximum Independant Set consiste à trouver un ensemble indépendant d'un graphe de cardinal maximum.*

L'algorithme de retour sur trace représente l'ensemble des solutions sous la forme d'un arbre, où chaque nœud de niveau i représente le choix pour le sommet i (est-ce qu'on le place ou non dans notre ensemble). Dans le pire des cas, l'arbre obtenu est de taille $2^{|V|}$.

Définition 12.9 (K-coloration) *Étant donné $G = (V, E)$ et $K > 0$, le problème de K-coloration consiste à trouver $c : V \to \{1, ..., K\}$ tel que pour tout $ij \in E$, $c(i) \neq c(j)$.*

Théorème 12.2 *L'algorithme de retour sur trace pour le problème de K-coloration s'exécute en $\mathcal{O}(1)$ opérations élémentaires en moyenne.*

Leçon 13

Algorithmes d'ordonnancement de tâches et de gestion de ressources.

Auteur·e·s: Sorci Émile, Bertrand Jules
Niveau : MPI
Pré-requis : Notions d'algorithmiques, $\mathbf{NP}$-complétude et approximations
Références : [Tanenbaum and Bos, 2014], [Benoit et al., 2013], [Casanova et al., 2008]

Introduction. Le but de cette leçon est de présenter l'ordonnancement de tâches d'un point de vue algorithmique tout d'abord. La question de l'ordonnancement permettront d'introduire la gestion de la concurrence et de la synchronisation. On finira par aborder l'ordonnancement au sein d'un système d'exploitation.

I Problème d'ordonnancement abstrait

De manière informelle, un ordonnancement de tâche consiste à répartir des tâches sur un ou plusieurs processeurs au cours du temps.

A Un premier problème

On considère un gymnase dans lequel on souhaite organiser n événements (s_i, e_i) (chaque événement est caractérisé par une date de début et une date de fin). Le but est de maximiser le nombre d'événements qui auront bien lieu dans le gymnase.

— choix 1 : l'événement qui minimise la durée $(e_i - s_i)$;
— choix 2 : l'événement qui a commence le plus tôt (qui minimise s_i) ;
— choix 3 : l'événement qui termine le plus tôt (qui minimise e_i).

Chacun des choix mène à un algorithme glouton différent.

Exercice 13.1 *Montrer que les algorithmes gloutons faisant le choix 1 et 2 ne sont pas optimaux.*

B Formalisation et ordonnancement de tâche indépendantes

Définition 13.1 *Soient n tâches $\mathcal{T} = \{T_1, ...T_n\}$ caractérisées par leurs temps d'exécution $w_1, ..., w_n$. Un* **ordonnancement** *de $\mathcal{T}$ est une fonction*

$$\sigma : \mathcal{T} \to \mathbb{N} \times P$$

où P est un ensemble de processeurs. Pour tout $T \in \mathcal{T}$, si $\sigma(T) = (t, p)$, alors la tâche T sera exécutée à l'instant t sur le processeur p.

On considère ici des processeurs identiques et on cherche à minimiser le maximum des temps d'exécution sur l'ensemble des processeurs. De plus, deux tâches sont dites indépendantes si elles peuvent être exécutées dans n'importe quel ordre.

Définition 13.2 ($\mathbf{INDEP}(p)$) *Étant donné un ensemble de tâche indépendantes $\mathcal{T}$ et un nombre de processeurs p, trouver un ordonnancement de temps d'exécution minimal.*

Théorème 13.1 *$\mathbf{INDEP}(p)$ est $\mathbf{NP}$-complet pour tout $p \geq 2$.*

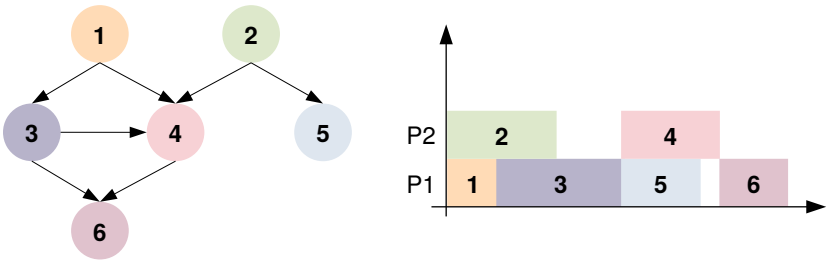
Exercice 13.2 *Donner un algorithme de programmation dynamique permettant de résoudre $\mathbf{INDEP}(2)$. Donner sa complexité.*

Malgré le théorème précédent, on peut trouver des algorithmes efficaces pour résoudre ce problème.
Développement 20. Algorithme d'approximation pour $\mathbf{INDEP}(p)$.

C Ordonnancement de tâches dépendantes

Définition 13.3 *Étant donné un ensemble de tâche $\mathcal{T}$, un* **graphe de tâches** *$G = (\mathcal{T}, E)$ est un ensemble de contraintes sur l'ordre d'exécution. Ainsi, si $T_iT_j \in E$, alors la tâche T_i soit être exécutée avant la tâche T_j.*

Exemple 13.1



Comme on peut s'en douter, la généralisation du problème précédent aux tâches dépendantes restent $\mathbf{NP}$-complet. Il existe cependant des heuristiques permettant d'obtenir de bons résultats, comme la **méthode du chemin critique**.

Exercice 13.3 *Donner une borne inférieur sur le temps d'exécution optimal d'un ordonnancement en fonction de la topologie du graphe de tâche.*

D Système d'exploitation et ordonnancement online

Dans le cadre d'un système d'exploitation, l'ensemble de tâches n'est pas connu à l'avance. La modélisation précédente ne s'applique donc pas dans ce cas. En particulier, les algorithmes online sont beaucoup proches de la réalité dans ce cadre.

Exercice 13.4 *Montrer que l'algorithme online qui exécute les tâches sur un unique processeur dans leur ordre d'arrivé minimise le maximum des temps de réponse de ces tâches.*

Une autre simplification concerne la capacité de nos tâches à s'exécuter en parallèle. Si celles-ci demandent l'accès à une ressource commune, des problème de synchronisation apparaissent.

II Concurrence et synchronisation

A Processus et Threads

Définition 13.4 *Un* **processus** *est une abstraction par le système d'un programme en cours d'exécution.*
Un **thread** *ou* **fil d'exécution** *d'un processus est une exécution (état des registres et de la pile) pour un processus.*

Exemple 13.2 *Il y a au moins un processus par fenêtre ouverte.*

Remarque 13.1 *Il existe de nombreux processus dont on n'a pas conscience, qui tourne continuellement en fond. Pour l'instant, on pourra considérer que tous les processus s'exécute en parallèle.*

Implémentation. Chaque processus dispose de son propre état des registres de l'UC, de son propre espace d'adressage et de ses propres fichiers ouverts. Ces informations sont gardés en mémoire dans la **table des processus**. De la même manière, les informations relatives à un thread sont stockées dans une table des threads.

Remarque 13.2 *L'espace d'adressage entre threads d'un même processus est le même, ils partagent les variables globales. Les piles et états des registres sont cependant distincts.*

Manipulation. La librairie `pthread` en C permet de manipuler les threads.

Exercice 13.5 (TP)

1. *Manipulation des threads en C.*
2. *Écrire un programme où plusieurs threads se comptent en incrémentant un compteur partagé.*

Le deuxième exercice montre un non déterminisme : il y a besoin d'exclusion mutuelle.

B Cohérence et synchronisation de processus

Section critique et exclusion mutuelle

Définition 13.5 *Une* **condition de concurrence** *est une situation où deux threads ou processus exécutent une zone du code d'écriture ou de lecture dans sur une mémoire partagée et dont le résultat dépend de l'ordre d'exécution des instructions. Une telle zone de code est appelée* **section critique**.
Si plusieurs threads n'entrent jamais dans leur section critique en même temps, on dit qu'il y a **exclusion mutuelle**.

Exemple 13.3 *Dans le programme précédent où plusieurs threads se comptent, la section critique correspond à la ligne d'incrémentation du compteur. On peut alors identifier la condition de concurrence.*

Implémentation de l'exclusion mutuelle par attente active

Premier essai Une première solution consiste à utiliser une variable globale verrou que l'on incrémente pour entrer en section critique. On peut ici noter un problème d'atomicité de l'opération qui empêche d'avoir vraiment exclusion mutuelle.

Solution de Peterson L'algorithme de Peterson permet de résoudre le problème de l'exclusion mutuelle pour 2 threads.

Solution de Lamport L'algorithme de la boulangerie de Lamport permet de résoudre le problème de l'exclusion mutuelle pour p threads.

Développement 26. On montre certaines propriété qu'assure l'algorithme de Peterson.

Ces deux solutions présentent un problème majeur : il s'agit d'attente active. Le processeur exécute du code qui ne fait rien, et donc de la ressource est perdue.

Implémentation de l'exclusion mutuelle avec des mutex et sémaphores

Définition 13.6 *Un* **sémaphore** *est une variable qui stocke un entier qui représente le nombre de ressources d'un type disponible. Deux opérations sont possibles sur un tel entier : le prendre (décrémenter) s'il est non nul (sinon, on bloque tant qu'il est nul) ou le libérer de manière réciproque. Ces opérations sont* **atomiques**.

Un **mutex** *est un sémaphore binaire.*

Pour implémenter les sémaphores et mutex dans le système, on utilise des interruptions. Lorsqu'un thread bloque, il est placé dans un état bloqué special. L'ordonnanceur (cf partie suivante) n'essaye plus de l'exécuter tant qu'il n'a pas récupérer l'information disant que le mutex ou sémaphore a été libéré. La librairie `pthread` permet d'utiliser des mutex, et la librairie `semaphore` les sémaphores.

III L'ordonnancement dans un système d'exploitation

A Le rôle de l'ordonnanceur

Le premier rôle de l'ordonnanceur consiste à répartir les processus et threads sur le processeur de sorte à ce que l'utilisateur ait une sortie fluide. Il assure alors une illusion de parallélisme, que l'on appelle pseudo-parallélisme. De plus, il gère l'état des threads en fonction des Entrées/Sorties, des mutex et des sémaphores. Enfin, il permet aussi de gérer le parallélisme réel (comme dans un processeur multi-cœurs par exemple).

B Stratégies d'ordonnancement sur un processeur.

On s'intéresse ici aux **online**, c'est-à-dire que l'on reçoit au fur et à mesure de nouvelles requêtes.

On commence par étudier des **stratégies non préemptives**, c'est-à-dire que le système ne peut pas arrêter un thread pendant son exécution.

Stratégie de la tâche la première arrivée. Une **tâche** est le temps entre deux Entrées/Sorties pour un thread. Cette stratégie consiste à exécuter les tâches dans leur ordre d'arrivé. On peut montrer un certain résultat d'optimalité pour cette stratégie (cf exercice 13.4).

Stratégie de la tâche la plus courte en premier. cette stratégie minimise le temps de réponse réponse mais, pour l'appliquer, il faut connaître le temps des tâches. En général ce n'est pas le cas, on utilise l'estimation par le calcul du **vieillissement**.

On étudie maintenant des **stratégies préemptives**, c'est-à-dire le système peut arrêter un thread à tout moment et le relancer au même endroit.

Stratégie du tourniquet (Round-Robin). On définit un **quantum** de temps sur lequel les threads sont s'exécuter avant d'être préempté. On utilise alors une structure de file pour savoir quel thread exécuter. La taille du quantum est souvent un compromis entre le coût de changement de contexte et du temps de réponse.

Ordonnancement par priorités. On a n groupes de priorités, on exécute les threads de priorité maximale d'abord selon une stratégie du tourniquet, puis, on passe à la suivante, etc. On peut alors avoir des priorités statistiques ou dynamiques qui dépendent du temps des tâches et la durée d'attente par exemple.

Leçon 14

Gestion et coordination de multiples fils d'exécution.

Auteur·e·s: Sorci Émile
Niveau : L2-L3
Pré-requis : Notions d'architecture, Notions d'algorithmiques
Références : [Tanenbaum and Bos, 2014], [Patterson and Hennessy, 2017]

I Processus et Threads

Motivation Si on veut exécuter plusieurs programmes en même temps, on a besoin d'une notion de parallélisme, ou de pseudo-parallélisme.

A Les processus

Définition 14.1 *Un* **processus** *est une abstraction par le système d'un programme en cours d'exécution.*

Exemple 14.1 *Il y a au moins un processus par fenêtre ouverte.*

Remarque 14.1 *Il existe de nombreux processus dont on n'a pas conscience, qui tourne continuellement en fond. Pour l'instant, on pourra considérer que tous les processus s'exécute en parallèle.*

Implémentation. Chaque processus dispose de son propre état des registres de l'UC, de son propre espace d'adressage et de ses propres fichiers ouverts. Ces informations sont gardés en mémoire dans la **table des processus**.

Manipulation. On peut manipuler ces processus via des appels sytèmes comme `fork`, `execve`, `exit`, `waitpid`, `kill`, `pipe`, `mmap`, etc.

B Les threads

Il peut arriver au sein d'une même application de vouloir exécuter plusieurs tâches en parallèle.

Définition 14.2 *Un* **thread** *ou* **fil d'exécution** *d'un processus est une exécution (état des registres et de la pile) pour un processus.*

Remarque 14.2 *L'espace d'adressage entre threads d'un même processus est le même, ils partagent les variables globales. Les piles et états des registres sont cependant distincts.*

Implémentation. Comme pour les processus, les informations relatives à un thread sont stockées dans une table des threads.

Manipulation. La librairie `pthread` en C permet de manipuler les threads.

Exercice 14.1 (TP)

1. *Manipulation des processus et threads en C pour voir la différence.*
2. *Écrire un programme où plusieurs threads se comptent en incrémentant un compteur partagé.*

Le deuxième exercice montre un non déterminisme : il y a besoin d'exclusion mutuelle.

II Cohérence et synchronisation entre threads

A Section critique et exclusion mutuelle

Définition 14.3 *Une* **condition de concurrence** *est une situation où deux threads ou processus exécutent une zone du code d'écriture ou de lecture dans sur une mémoire partagée et dont le résultat dépend de l'ordre d'exécution des instructions. Une telle zone de code est appelée* **section critique**.

Si plusieurs threads n'entrent jamais dans leur section critique en même temps, on dit qu'il y a **exclusion mutuelle**.

Exemple 14.2 *Dans le programme précédent où plusieurs threads se comptent, la section critique correspond à la ligne d'incrémentation du compteur. On peut alors identifier la condition de concurrence.*

B Implémentation de l'exclusion mutuelle par attente active

Premier essai. Une première solution consiste à utiliser une variable globale verrou que l'on incrémente pour entrer en section critique. On peut ici noter un problème d'atomicité de l'opération qui empêche d'avoir vraiment exclusion mutuelle.

Solution de Peterson. L'algorithme de Peterson permet de résoudre le problème de l'exclusion mutuelle pour 2 threads.
Développement 26. On montre certaines propriété qu'assure l'algorithme de Peterson.

Solution de Lamport. L'algorithme de la boulangerie de Lamport permet de résoudre le problème de l'exclusion mutuelle pour p threads.

Ces deux solutions présentent un problème majeur : il s'agit d'attente active. Le processeur exécute du code qui ne fait rien, et donc de la ressource est perdue.

C Implémentation de l'exclusion mutuelle avec des mutex et sémaphores

Définition 14.4 *Un* **sémaphore** *est une variable qui stocke un entier qui représente le nombre de ressources d'un type disponible. Deux opérations sont possibles sur un tel entier : le prendre (décrémenter) s'il est non nul (sinon, on bloque tant qu'il est nul) ou le libérer de manière réciproque. Ces opérations sont* **atomiques**.

Un **mutex** *est un sémaphore binaire.*

Implémentation en système. On utilise des interruptions. Lorsqu'un thread bloque, il est placé dans un état bloqué special. L'ordonnanceur (cf partie suivante) n'essaye plus de l'exécuter tant qu'il n'a pas récupérer l'information disant que le mutex ou sémaphore a été libéré.

Manipulation. La librairie `pthread` permet d'utiliser des mutex, et la librairie `semaphore` les sémaphores.

D Exercices

Exercice 14.2

1. *Implémentation d'un produit de matrice en parallèle.*
2. *Implémentation d'un système où un thread lit des requêtes dans un fichier et les fait exécuter par d'autres threads.*
3. *Implémentation de la solution de Dijkstra au problème du dîner des philosophes.*
4. *Résolution du problème des Rédacteurs-Lecteurs.*

III L'ordonnancement des threads

A Le rôle de l'ordonnanceur

Le premier rôle de l'ordonnanceur consiste à répartir les threads sur le processeur de sorte à ce que l'utilisateur ait une sortie fluide. Il assure alors une illusion de parallélisme, que l'on appelle pseudo-parallélisme. De plus, il gère l'état des threads en fonction des Entrées/Sorties, des mutex et des sémaphores. Enfin, il permet aussi de gérer le parallélisme réel (comme dans un processeur multi-cœurs par exemple).

B Stratégies d'ordonnancement sur un processeur.

On s'intéresse ici aux **online**, c'est-à-dire que l'on reçoit au fur et à mesure de nouvelles requêtes.

> **Remarque 14.3** *Les stratégies offline sont étudiées dans d'autres cadres.*

On commence par étudier des **stratégies non préemptives**, c'est-à-dire que le système ne peut pas arrêter un thread pendant son exécution.

Stratégie de la tâche la première arrivée. Une **tâche** est le temps entre deux Entrées/Sorties pour un thread. Cette stratégie consiste à exécuter les tâches dans leur ordre d'arrivé. On peut montrer un certain résultat d'optimalité pour cette stratégie.
Développement 27. Étude d'un algorithme d'ordonnancement online.

Stratégie de la tâche la plus courte en premier. cette stratégie minimise le temps de réponse réponse mais, pour l'appliquer, il faut connaître le temps des tâches. En général ce n'est pas le cas, on utilise l'estimation par le calcul du **vieillissement**.

On étudie maintenant des **stratégies préemptives**, c'est-à-dire le système peut arrêter un thread à tout moment et le relancer au même endroit.

Stratégie du tourniquet (Round-Robin). On définit un **quantum** de temps sur lequel les threads sont s'exécuter avant d'être préempté. On utilise alors une structure de file pour savoir quel thread exécuter. La taille du quantum est souvent un compromis entre le coût de changement de contexte et du temps de réponse.

Ordonnancement par priorités. On a n groupes de priorités, on exécute les threads de priorité maximale d'abord selon une stratégie du tourniquet, puis, on passe à la suivante, etc. On peut alors avoir des priorités statistiques ou dynamiques qui dépendent du temps des tâches et la durée d'attente par exemple.

IV Le coût du parallélisme

Loi d'Amdall. Si un programme séquentiel s'exécute en temps $T = (1-p)T + pT$ où $(1-p)$ est la proportion de temps qui ne peut pas être parallélisé (E/S par exemple). On a alors, en utilisant s thread, un temps d'exécution T_s qui vérifie

$$T_s \geq (1-p)T + p\frac{T}{s}$$

Tout n'est pas parallélisable et paralléliser ne divise pas le temps d'exécution par le nombre de threads.

Coût des changements de contexte. Les changements de contexte sont coûteux, il n'est pas souhaitable qu'une part importante du temps y soit affectée.

Risques d'interblocages. Plus il y a de fils d'exécutions concurrents voulant accéder à des ressources communes, plus il y a de risques d'**interblocages**. Un interblocage est une situation durant laquelle plusieurs fils d'exécution ont besoin d'une ressource détenue par une autre : ils sont alors tous en attente. Dans un système classique, il est très difficile de les éviter et d'en sortir ; la plupart du temps, une politique de l'autruche est appliquée.

Programmation et preuves. La programmation avec plusieurs fils d'exécution est évidemment plus complexe qu'une manière plus classique. De la même manière, montrer la correction du programme est plus difficile.

Leçon 15

Hiérarchie mémoire. Structure et performances.

Auteur·e·s: Marin Malory
Niveau : L3
Pré-requis : Base architecture des ordinateurs, algorithmiques
Références : [Patterson and Hennessy, 2017]

I Introduction

Un ordinateur contient plusieurs niveaux de mémoires : de la mémoire cache rapide proche du processeur, de la mémoire RAM moyennement rapide et de la mémoire morte d'accès lent pour des raisons économiques.

Lorsqu'un **processus** (programme en cours d'exécution) demande une donnée, comment faire en sorte qu'elle arrive au processeur de manière efficace ?

Pour organiser notre mémoire, on se base sur deux principes de localité :

Définition 15.1 (Localité temporelle) *Lorsqu'un processus utilise une donnée, il a tendance à la ré-utiliser peu de temps après.*

Définition 15.2 (Localité spatiale) *Lorsqu'un processus utilise une donnée, il a tendance à aussi utiliser d'autres données proches en mémoire de celle-ci.*

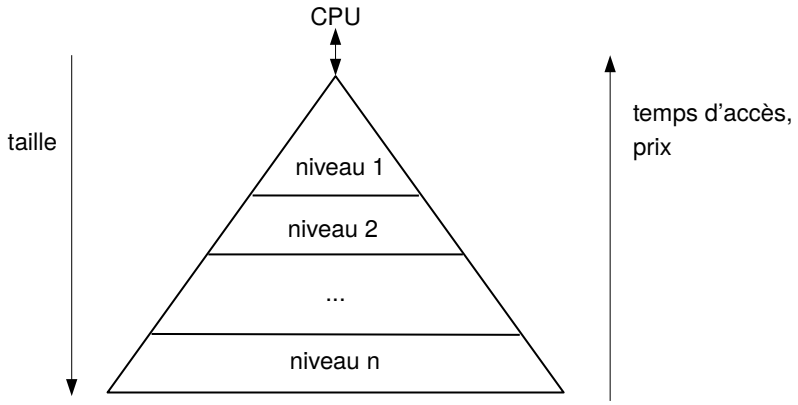
On organise alors la mémoire d'un ordinateur en **hiérarchie mémoire**. Une telle organisation consiste en un empilement de mémoires de différentes tailles et vitesses (qu'on numérote de 1 à n).

Plus une donnée est utilisée souvent, plus elle est proche du processeur (utilisation de la localité temporelle).

Lorsqu'un processus demande une donnée x :

— (**hit**) si la donnée x est présente au niveau k , alors la donnée est transférée au niveau $k-1$ par bloc (utilisation de la localité spatiale),

— (**miss**) sinon, la donnée est demandée au niveau $k+1$.



Entre la mémoire principale et le processeur, les niveaux de mémoires sont appelées **caches**. Même si une hiérarchie mémoire peut contenir plusieurs niveaux, les données sont copiées seulement entre deux niveaux à la fois, on se concentrera sur seulement deux niveaux, l'un rapide et l'autre lente.

Définitions. L'unité d'information minimale pouvant être présente ou non dans une hiérarchie mémoire à deux niveaux est appelée **ligne de cache**. On appelle **taux de défaut mémoire** la fraction de mémoire accédée qui n'est pas trouvée en mémoire rapide. Le **coût de l'échec** est le temps nécessaire pour remplacer une ligne en mémoire rapide par une ligne en mémoire lente.

Technologies mémoires. Il y a quatre technologies de mémoires principalement utilisées.

1. les DRAM (*dynamic random access memory*) pour la mémoire principale ;
2. les SRAM (*static random access memory*) pour les niveaux plus proches du processeur ;
3. la mémoire flash, pour la mémoire secondaire des appareils mobiles ;
4. les disques magnétiques pour le niveau le plus grand et lent d'une hiérarchie pour un serveur.

Les programmes favorisent une partie de leurs adresses à tout instant par localité temporelle (tendance à réutiliser une donnée récente) et par localité spatiale (tendance à utiliser une donnée proche déjà utilisée). La hiérarchie mémoire utilise à son avantage la localité temporelle en gardant proche du processeur les données récemment utilisées ; et de la localité spatiale en mouvant les données par blocs contiguës.

Structure de données. Il est important pour le programmeur de connaître l'importance de cette hiérarchie mémoire dans son choix de structures de données par exemple.
Développement 3. Les **B-arbres** sont un exemple de structure de donnée prenant en compte cette hiérarchie.

II Introduction à la mémoire cache

On commence ici par considérer un cache simple, où les requêtes et les ligne de cache correspondent à un unique mot (32 bits).

Deux problématiques apparaissent directement lorsqu'on essaye de concevoir un cache : comment savoir si un mot se trouve dans le cache et s'il y est, comment le trouver ?

Définition 15.3 *Un* **cache à acces direct** *est un cache pour lequel chaque bloc de la mémoire est associé à une unique adresse dans le cache.*

En général, si on essaye de placer une ligne x dans un cache à N blocs, on la placera à l'adresse $x \mod N$.

Puisque cette fonction n'est pas bijective, pour savoir si un mot à une certaine adresse du cache est le bon, on ajoute des **étiquettes** (*tags*). De plus, on ajoute un **bit de validité** permettant de savoir si l'information présente en cache est correcte ou non (au démarrage par exemple).

Exercice 15.1 *Combien de bits sont nécessaires pour un cache à accès direct pour une mémoire de 16KiO de données et des mots blocs de 4 mots, chaque adresse étant sur 64bits.*

Remarque 15.1 *La taille d'un ligne de cache est une métrique importante. Une grande taille exploite mieux la localité spatiale, mais augmente le nombre de cache miss.*

A Écriture dans un cache

On considère le cas où le processeur exécute une instruction de stockage. Si on écrit seulement dans le cache, la mémoire n'est plus **cohérente** avec le cache. Une première solution consiste à toujours écrire dans le cache et dans la mémoire : on parle de politique **write-through**.

Cette politique cause évidemment des problèmes de performance. Une autre politique, le **write-back**, consiste à écrire en mémoire la ligne seulement lorsqu'elle est remplacée. Cette politique est plus efficace, mais plus dure à implanter.

III Performance et optimisation de cache

A Mesure de performance

Dans cette partie, on donne des métriques permettant de mesurer la performance d'un cache.

— **Temps CPU** : temps de calcul du CPU, mesuré en cycles d'horloges ou en seconde ;

— T_H : période de l'horloge (seconde par cycle) ;

— **Cycles Exécution** : nombre de cycle d'exécution du CPU ;

— **Cycles Mémoire** : nombre de cycle où le CPU attend de la mémoire (principalement dû aux caches miss).

On a alors

$$\textbf{Temps CPU} = (\textbf{Cycles Exécution} + \textbf{Cycles Mémoire}) \times T_H$$

On s'intéresse maintenant à la valeur **Cycles Mémoire**, que l'on veut minimiser. Ce paramètre dépend de deux paramètres : le nombre de cache miss et la pénalité associé à chaque cache miss.

B Optimisation : cache associatif

Pour réduire le nombre de cache miss, on peut revenir sur l'idée d'un cache à accès direct.

Définition 15.4 *Un cache* **pleinement associatif** *est un cache dans lequel chaque ligne peut être placée n'importe où dans le cache.*

Il existe aussi un entre deux : les caches associatifs par paquet. De nouvelles problématiques apparaissent avec ce type de cache. En effet, dans un tel cache, toutes les entrées du cache doivent être parcourues pour trouver une ligne. Pour améliorer la performance, cette recherche est parallélisée, mais cela augmente le coût matériel.

Choix de la ligne à remplacer. Lors d'un cache miss, cette fois on peut choisir la ligne de cache à évincer. Il existe plusieurs stratégies :

— FIFO : on retire la ligne de cache qui est rentrée en premier en mémoire principale ;

— LRU (Least Recently Used) : on retire la ligne qui a été utilisé la moins récemment .

On parle d'algorithme de pagination.

Développement 21. Comment mesurer la performance de tel algorithme online ?

C Optimisation : plusieurs niveaux de caches

Afin de réduire le cache penalty, on peut superposer plusieurs niveaux de caches.

IV Mémoire virtuelle

On reprend les concepts vu précédemment, avec cette fois pour mémoire rapide la mémoire principale, et pour mémoire lente la mémoire morte.

A Principes

Lorsqu'il y a plusieurs processus (programme en cours d'exécution), on aimerait que chacun ait l'impression de s'exécuter tout seul de manière sécurisé (protection), et que chacun ait accès à l'entièreté de la mémoire (espace).

Ces problématiques ont motivé la **mémoire virtuelle**, qui est une technique utilisant la mémoire principale comme un ≪ cache ≫ pour la mémoire secondaire (ou mémoire morte).

Protection. Dans l'idéal, chaque processus a son propre espace d'adressage, et la mémoire virtuelle est chargé de traduire l'adresse du processus à une **adresse physique** dans la mémoire principale. Cette traduction assure de la **protection**.

Définition 15.5 (Protection) *La protection est un ensemble de mécanisme permettant de s'assurer que plusieurs processus, partageant le processeur, la mémoire ou les I/O, ne peuvent pas lire ou écrire dans les données des autres.*

Ces mécanismes permettent aussi d'isoler l'OS des processus utilisateurs.

Espace. Historiquement, lorsqu'un programme devenait trop grand pour tenir en mémoire centrale, c'était au programmeur de le diviser en tâches indépendantes pour le faire rentrer. La mémoire virtuelle permet d'alléger ce fardeau et de gérer les deux niveaux de hiérarchie mémoire : la mémoire principale (aussi appelée **mémoire physique** pour séparer de la mémoire virtuelle) et la mémoire secondaire.

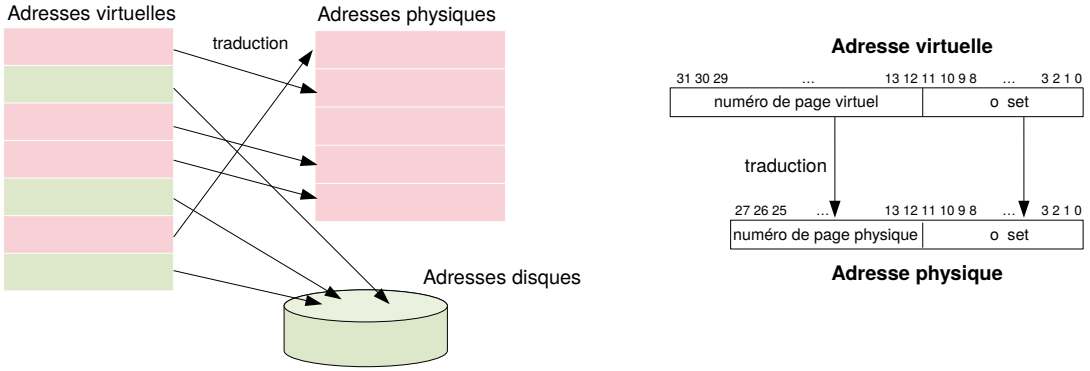
B Adresses et page

On appelle **page** un bloc mémoire de la mémoire virtuelle, et un **page fault** l'évènement qui a lieu lorsque le processeur veut accéder à une page non présente en mémoire principale.

Adresse virtuelle et adresse physique. Avec la mémoire virtuelle, le processeur produit une **adresse virtuelle**, qui est traduite (via une combinaison de *hardware* et de *software*) en une adresse physique utilisable pour accéder à la mémoire principale. On appelle ce mécanisme la **traduction d'adresse**.

Remarque 15.2 *La partie matérielle chargé de ces traductions (entre autres) est l'unité de gestion de mémoire (MMU). La partie logicielle est le gestionnaire de mémoire, faisant partie du système d'exploitation.*

Dans la mémoire virtuelle, l'adresse est divisé en un *numéro de page virtuel* ainsi qu'un *offset*. Le nombre de bits dans l'offset détermine la taille de la page.

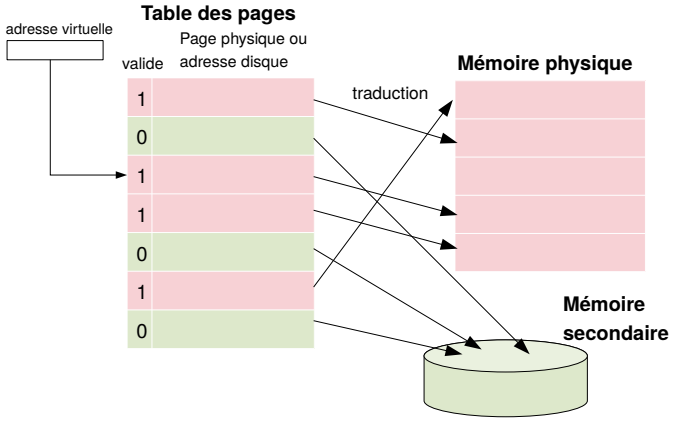


Puisqu'un page fault a un coût énorme, on cherche à optimiser le placement des pages. En particulier, en cas de page fault, la page virtuelle peut être associée à n'importe quelle adresse physique : c'est exactement le même principe qu'un cache associatif.

C Placement et remplacement des pages

Placement. Dans les systèmes de mémoires virtuels, on localise les pages via une **table des pages**.

Définition 15.6 (Table des pages) *Table contenant la traduction des adresses virtuelles vers les adresses physiques dans un système de mémoire virtuelle. Cette table, stockée en mémoire principale, est le plus souvent indexés par les adresses virtuelles et contient les adresses physiques, ainsi qu'un bit de validité pour chaque page.*



Pour indiquer la localisation de cette table des pages en mémoire, le hardware contient un registre spécial, appelé **registre de la table des pages**.

Remarque 15.3 *La table des pages, avec le compteur ordinal et les registres définissent* **le contexte du processus**. *Lorsque l'OS décide de donner la main à un autre processus, il doit changer le contexte et change notamment la valeur du registre de la table des pages. L'OS est aussi chargé de mettre à jour cette table si besoin.*

Remarque 15.4 *Pour chaque processus, il existe un espace en mémoire secondaire, appelé* **mémoire swap**, *permettant de garder les pages virtuelles qui ne sont pas en mémoire principale.*

Remplacement de pages. Lorsqu'il y a un page fault, c'est-à-dire que le processus essaye d'accéder à une adresse virtuelle dont le bit de validité est 0, l'OS prend la main (via exception). L'OS va alors chercher la page dans le prochain niveau de la hiérarchie (mémoire secondaire), et décide de placer la page en mémoire principale. Le problème est alors le même que pour un cache miss dans un cache pleinement associatif : des algorithmes du type LRU peuvent être utilisés.

Leçon 16

Mémoire : du bit à l'abstraction vue par les processus.

Auteur·e·s: Marin Malory
Niveau : L3
Pré-requis : Circuits combinatoires
Références : [Patterson and Hennessy, 2017]

I Circuits séquentiels

Avec des portes logiques, on peut construire des **circuits combinatoires** permettant de calculer une fonction booléenne. On va maintenant inclure des mémoires dans ces circuits.

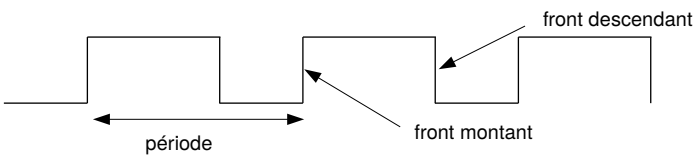
Définition 16.1 (Logique séquentielle) *Un* **circuit séquentiel** *est en groupe d'éléments logiques contenant de la mémoire. Ainsi, la fonction calculée par un tel système dépend des entrées mais aussi de l'état actuel de sa mémoire.*

A Horloge

Avant de discuter des circuits séquentiels, il est utile de discuter la notion d'horloge. On parle alors d'**élément à état** pour un élément contenant de la mémoire.

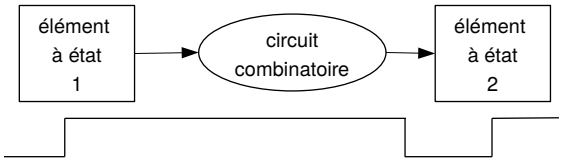
Pourquoi avoir une horloge ? Une horloge est nécessaire lorsque un système contient des états devant être mis à jour.

Définition 16.2 (Horloge) *Une* **horloge** *est un signal libre ayant une* **période** *(et donc une* **fréquence***) fixée. Une période d'horloge est appelée* **cycle d'horloge**.

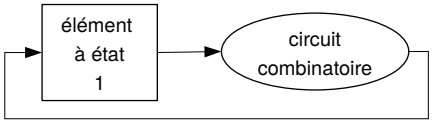


L'état d'un élément à état est mis à jour au **front montant** de l'horloge, et lorsqu'un système contient une unique horloge, on dit qu'il est **synchrone**.

Stabilité. Entre deux front montants d'horloge, un élément à état peut être mis à jour et une contrainte apparaît : à chaque front montant, tout les éléments à états doivent être **valide** et **stable**. Cela pose une contrainte sur la fréquence de l'horloge.



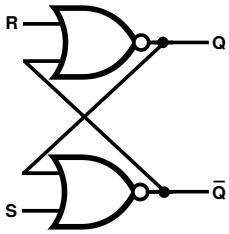
L'entrée d'un circuit combinatoire via d'un élément en état et la sortie est écrite dans un élément à mémoire. Les états sont mis à jours aux fronts montants.



On peut mettre à jour un élément à état via un circuit combinatoire en un seul cycle d'horloge.

B Éléments de mémoire : verrous, registres et mémoire adressable

Un premier élément à état. Les éléments de mémoire les plus simple n'ont pas de signal d'horloge en entrée. Un exemple de tel élément est le **verrou S-R** (S pour set, R pour reset).



Q et $\overline{Q}$ représente la valeur de l'état du verrou.

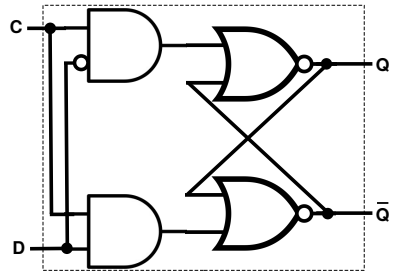
- Lorsque R et S sont à 0, Q est l'état précédent, et la seconde porte NOR agit comme un inverseur.
- Lorsque S vaut 1, on a $\overline{Q}$ vaut 0 et donc Q vaut 1.
- En mettant R à 1, on remet l'état à 0.
- Mettre S et R à 1 peut causer une oscillation et donc une perte de stabilité.

Cet élément sert de base pour construire des éléments mémoires pouvant être mis à jour via l'horloge.

Registres et verrous. Les **registres** (ou **flip-flop**, ou **bascule**) et les **verrous** dont les éléments mémoires les plus simples. Dans les deux éléments, la sortie est égal à la valeur stocké dans l'état, la différence entre les deux venant du moment où l'état est mis à jour.

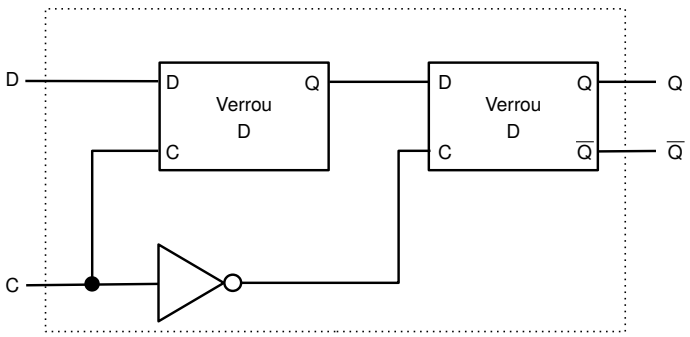
Dans un registre, le changement d'état est déclenché par l'horloge tandis que dans un verrou, il est déclenché par l'entrée.

On utilisera ici principalement une **bascule D**, que l'on peut construire via deux **verrous D**.



Le verrou R-S sert à stocker l'état du verrou. Au front montant (C vaut 1), on a un reset si D vaut 0, et sinon on fait un set.

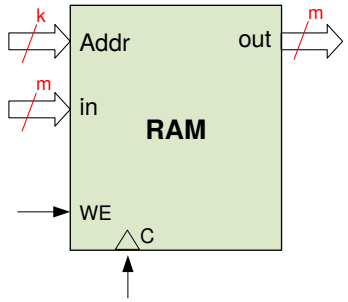
Lorsque l'horloge est basse, le premier verrou est passant et le second bloquant. Au front montant, le second devient passant et le premier bloque, on garde l'état D jusqu'au prochain front montant.



Exercice 16.1 *Dessiner les chronogrammes des éléments ci-dessus, en supposant que tous les états soient initialement à* 0.

Exercice 16.2 *Construire un verrou à partir d'un multiplexer 2 :1. Avec deux tels verrous, construire un registre.*

Mémoire adressable (RAM). Une mémoire adressable est composé d'un ensemble de registres qui peuvent être lus ou écris via une adresse d'entrée.



Exemple d'interface d'une RAM recevant une adresse sur k bits, une donnée d'entrée sur m bits, un bit Write-Enable ainsi que l'horloge, et renvoyant une donnée sur m bits.

Exercice 16.3 *Implanter l'interface ci-dessus en utilisant $2^k \times m$ bascules D, ainsi qu'un demultiplexer $k : 2^k$, ainsi qu'un multiplexer $2^k : m$.*

Remarque 16.1 *La mémoire adressable construite ci-dessus est* **statique**, *contrairement à une mémoire* **dynamique** *(qui n'utilise pas de flip-flop) qui a besoin d'être rafraîchit périodiquement. On fait alors la distinction entre une* **SRAM** *(plus rapide, plus chère) et une* **DRAM**

De la même manière qu'une fonction booléenne est l'objet théorique associé aux circuits combinatoire, il est possible de donner un penchant similaire pour les circuits séquentiels. L'équivalent théorique de ces derniers sont les **automates finis avec sortie**, tels que les machines de Moore ou de Mealy.

II Hiérarchie mémoire

Un ordinateur contient plusieurs niveaux de mémoires : de la mémoire cache rapide proche du processeur, de la mémoire RAM moyennement rapide et de la mémoire morte lente d'accès, principalement dû à des raisons économiques.

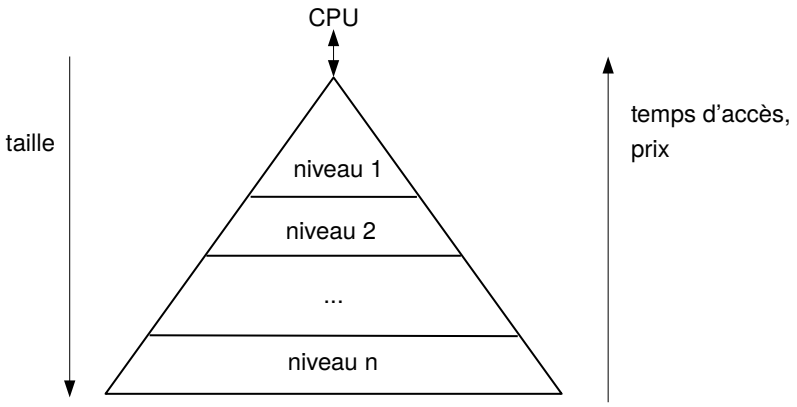
Lorsqu'un **processus** (programme en cours d'exécution) demande une donnée, comment faire en sorte qu'elle arrive au processeur de manière efficace ?

Pour organiser notre mémoire, on se base sur deux principes de localité :

Définition 16.3 (Localité temporelle) *Lorsqu'un processus utilise une donnée, il a tendance à la ré-utiliser pas longtemps après.*

Définition 16.4 (Localité spatiale) *Lorsqu'un processus utilise une donnée, il a tendance à aussi utiliser d'autres données proche en mémoire de celle-ci.*

On organise alors la mémoire d'un ordinateur en **hiérarchie mémoire**.



Plus une donnée est utilisée souvent, plus elle est proche du processeur (utilisation de la localité temporelle).

Lorsqu'un processus demande une donnée x :

— si la donnée x est présente au niveau k (**hit**), alors la donnée est transférée au niveau $k-1$ par bloc (utilisation de la localité spatiale),

— sinon (**miss**), la donnée est demandée au niveau $k+1$.

Entre la mémoire principale et le processeur, les niveaux de mémoires sont appelées **caches**.

Structure de données. Il est important pour le programmeur de connaître l'importance de cette hiérarchie mémoire dans son choix de structures de données par exemple.
Développement 3. Les **B-arbres** sont un exemple de structure de donnée prenant en compte cette hiérarchie.

III Processus et mémoire virtuelle

A Principes

Lorsqu'il y a plusieurs processus (programme en cours d'exécution), on aimerait que chacun ait l'impression de s'exécuter tout seul de manière sécurisé (protection), et que chacun ait accès à l'entièreté de la mémoire (espace).

Ces problématiques ont motivé la **mémoire virtuelle**, qui est une technique utilisant la mémoire principale comme un ≪ cache ≫ pour la mémoire secondaire.

Protection. Dans l'idéal, chaque processus a son propre espace d'adressage, et la mémoire virtuelle est chargé de traduire l'adresse du processus à une **adresse physique** dans la mémoire principale. Cette traduction assure de la **protection**.

Définition 16.5 (Protection) *La protection est un ensemble de mécanisme permettant de s'assurer que plusieurs processus, partageant le processeur, la mémoire ou les I/O, ne peuvent pas lire ou écrire dans les données des autres.*

Ces mécanismes permettent aussi d'isoler l'OS des processus utilisateurs.

Espace. Historiquement, lorsqu'un programme devenait trop grand pour tenir en mémoire centrale, c'était au programmeur de le diviser en tâches indépendantes pour le faire rentrer. La mémoire virtuelle permet d'alléger ce fardeau et de gérer les deux niveaux de hiérarchie mémoire : la mémoire principale (aussi appelée **mémoire physique** pour séparer de la mémoire virtuelle) et la mémoire secondaire.

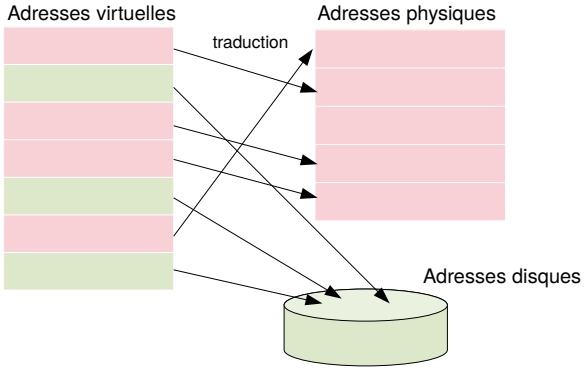
B Adresses et page

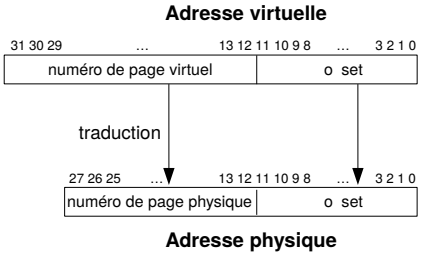
On appelle **page** un block mémoire de la mémoire virtuelle, et un **page fault** l'évènement qui a lieu lorsque le processeur veut accéder à une page non présente en mémoire principale.

Adresse virtuelle et adresse physique. Avec la mémoire virtuelle, le processeur produit une **adresse virtuelle**, qui est traduite (via une combinaison de *hardware* et de *software*) en une adresse physique utilisable pour accéder à la mémoire principale. On appelle ce mécanisme la **traduction d'adresse**.

> **Remarque 16.2** *La partie matérielle chargé de ces traductions (entre autres) est l'unité de gestion de mémoire (MMU). La partie logicielle est le gestionnaire de mémoire, faisant partie du système d'exploitation.*

Dans la mémoire virtuelle, l'adresse est divisé en un *numéro de page virtuel* ainsi qu'un *offset*. Le nombre de bits dans l'offset détermine la taille de la page.





> **Remarque 16.3** *De nos jours, les deux niveaux de mémoires utilisent des DRAMs et de la mémoire flash pour les appareil personnels, et des DRAMs et des disques magnétiques pour les serveurs.*

Relocalisation. L'espace d'adressage d'un processus est principalement constitué d'un pile (pour l'exécution) et d'un tas (pour la mémoire dynamique). La mémoire virtuelle permet de ne pas à avoir chercher un espace suffisant et contiguë en mémoire principale, mais seulement de chercher un nombre suffisant de pages.

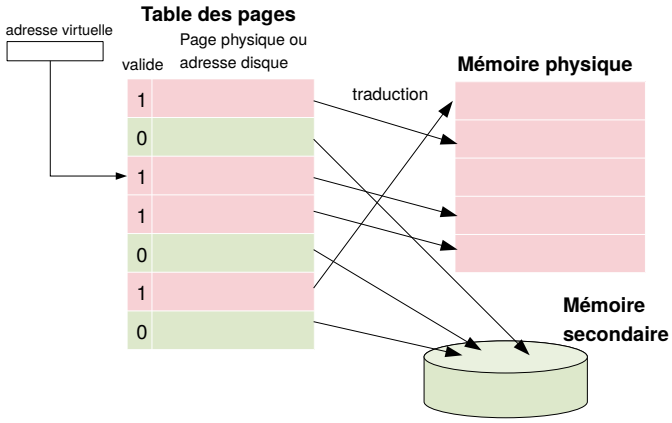
Puisqu'un page fault a un coût énorme, on cherche à optimiser le placement des pages. En particulier, en cas de page fault, la page virtuelle peut être associée à n'importe quelle adresse physique.

Le placement et le remplacement des pages en mémoire physique est alors assuré par le système d'exploitation. Même si un traitement logiciel peut paraître ici très coûteux, il reste négligeable face au coût d'un page fault.

C Placement et remplacement des pages

Placement. Dans les systèmes de mémoires virtuels, on localise les pages via une **table des pages**.

Définition 16.6 (Table des pages) *Table contenant la traduction des adresses virtuelles vers les adresses physiques dans un système de mémoire virtuelle. Cette table, stockée en mémoire principale, est le plus souvent indexés par les adresses virtuelles et contient les adresses physiques, ainsi qu'un bit de validité pour chaque page.*



Pour indiquer la localisation de cette table des pages en mémoire, le hardware contient un registre spécial, appelé **registre de la table des pages**.

Remarque 16.4 *La table des pages, avec le compteur ordinal et les registres définissent* **le contexte du processus**. *Lorsque l'OS décide de donner la main à un autre processus, il doit changer le contexte et change notamment la valeur du registre de la table des pages. L'OS est aussi chargé de mettre à jour cette table si besoin.*

Remarque 16.5 *Pour chaque processus, il existe un espace en mémoire secondaire, appelé* **mémoire swap**, *permettant de garder les pages virtuelles qui ne sont pas en mémoire principale.*

Remplacement de pages. Lorsqu'il y a un page fault, c'est-à-dire que le processus essaye d'accéder à une adresse virtuelle dont le bit de validité est 0, l'OS prend la main (via exception). L'OS va alors chercher la page dans le prochain niveau de la hiérarchie (mémoire secondaire), et décide de placer la page en mémoire principale.

Si la mémoire principale est déjà pleine, il faut choisir une page à remplacer. Il existe plusieurs stratégies :

— FIFO : on retire la page qui est rentré en premier en mémoire principale ;

— LRU (Least Recently Used) : on retire la page qui a été utilisé la moins récemment .

Développement 21. Comment mesurer la performance de tel algorithme online ?

Leçon 17

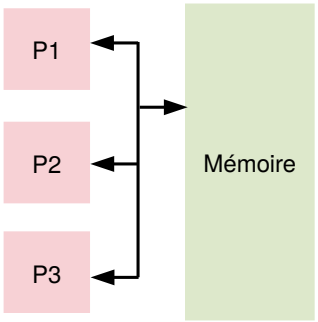
Problèmes et stratégies de cohérence et de synchronisation.

Auteur·e·s: Sorci Émile
Niveau : L3
Pré-requis : Notions d'architecture, hiérarchie mémoire
Références : [Tanenbaum and Bos, 2014], [Stallings, 2003]

I Introduction au parallélisme

A Un premier modèle simple

On présente ici un modèle plus sophistiqué qu'une architecture de Von Neuman : le modèle de la **mémoire partagée** entre plusieurs processeurs.



B Processus et threads

Définition 17.1 *Un* **processus** *est un programme en cours d'exécution, pouvant lui-même contenir plusieurs fils d'exécution appelés* **threads**.

Définition 17.2 *Chaque processus contient son propre espace d'adressage virtuel, un identifiant, un ensemble de fichiers ouverts et d'autres informations, toutes stockées dans une structure de donnée en mémoire appelée* **bloc de contrôle des processus**.

Deux threads d'un même processus partagent leur espace d'adressage. Ils ont cependant un **pointeur de pile** différent.

Remarque 17.1

— *Dans un système d'exploitation, on isole les processus à l'aide de la mémoire virtuelle pour des raisons de sécurité. Ce n'est pas le cas des threads car ils coopèrent entre eux.*

— *Les processus et threads ne servent pas qu'au parallélisme pur : ils permettent aussi une illusion du parallélisme.*

C Programmation avec des threads

Exercice 17.1 (TP)

1. *Introduction à la bibliothèque* `pthread`.
2. *Multiplication de matrice multi-threads : chaque threads calcul un ensemble de lignes. Il n'y a pas de problème de synchronisation dans ce cas.*
3. **Un premier exemple de problème de cohérence** *: les threads se comptent. On remarquera le non déterminisme et le besoin d'isoler certaines sections du code.*

II Cohérence et synchronisation entre threads

A Section critique et exclusion mutuelle

Définition 17.3 *Une* **condition de concurrence** *est une situation où deux threads ou processus exécutent une zone du code d'écriture ou de lecture dans sur une mémoire partagée et dont le résultat dépend de l'ordre d'exécution des instructions. Une telle zone de code est appelée* **section critique**.

Si plusieurs threads n'entrent jamais dans leur section critique en même temps, on dit qu'il y a **exclusion mutuelle**.

Exemple 17.1 *Dans le programme précédent où plusieurs threads se comptent, la section critique correspond à la ligne d'incrémentation du compteur. On peut alors identifier la condition de concurrence.*

B Implémentation de l'exclusion mutuelle par attente active

Premier essai. Une première solution consiste à utiliser une variable globale verrou que l'on incrémente pour entrer en section critique. On peut ici noter un problème d'atomicité de l'opération qui empêche d'avoir vraiment exclusion mutuelle.

Solution de Peterson. L'algorithme de Peterson permet de résoudre le problème de l'exclusion mutuelle pour 2 threads.
Développement 26. On montre certaines propriétés qu'assure l'algorithme de Peterson.

Solution de Lamport. L'algorithme de la boulangerie de Lamport permet de résoudre le problème de l'exclusion mutuelle pour p threads.

Ces deux solutions présentent un problème majeur : il s'agit d'attente active. Le processeur exécute du code qui ne fait rien, et donc de la ressource est perdue.

C Implémentation de l'exclusion mutuelle avec des mutex et sémaphores

Définition 17.4 *Un* **sémaphore** *est une variable qui stocke un entier qui représente le nombre de ressources d'un type disponible. Deux opérations sont possibles sur un tel entier : le prendre (décrémenter) s'il est non nul (sinon, on bloque tant qu'il est nul) ou le libérer de manière réciproque. Ces opérations*

sont **atomiques**.
Un **mutex** *est un sémaphore binaire.*

Implémentation en système. On utilise des interruptions. Lorsqu'un thread bloque, il est placé dans un état bloqué special. L'ordonnanceur (cf partie suivante) n'essaye plus de l'exécuter tant qu'il n'a pas récupérer l'information disant que le mutex ou sémaphore a été libéré.

Exemple 17.2 *On pourra illustrer l'utilisation des mutex avec le problème du dîner des philosophes.*

Exercice 17.2 (TP2)

1. *Introduction à l'utilisation de sémaphore et mutex.*
2. *Multiplication de matrice où plusieurs threads calculent sur le même coefficient en même temps.*
3. *Problème des producteurs et consommateurs.*

D Quelques mots sur l'ordonnancement

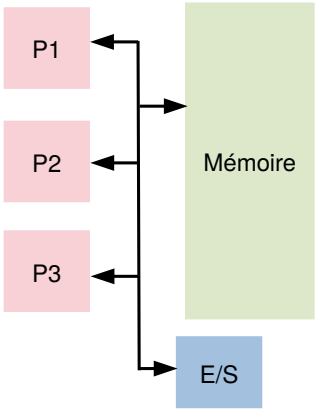
Définition 17.5 *Les processus et threads peuvent être dans plusieurs* **états** *dont bloqué, en attente et en exécution. C'est l'***ordonnanceur** *qui s'occupe de changer les états.*

Ainsi, l'ordonnanceur assure à la fois l'illusion du parallélisme en permettant à chaque processus de s'exécuter, mais aussi de la parallélisation réelle en répartissant les processus.

Stratégies. L'ordonnanceur peut avoir différentes stratégies : par exemple en exécutant le premier processus disponible, ou le plus court, ou en exécutant chaque processus les uns après les autres durant un certain quantum de temps (*Round-Robin*).
Développement 27. Étude d'un algorithme d'ordonnancement online.

III Les interblocages

On étend notre modèle multi-processeurs en intégrant les Entrées/Sorties comme des **ressources** auxquelles les threads peuvent accéder de façon exclusive.



A Notion d'interblocage

Définition 17.6 *Un ensemble de processus est en interblocage si chaque processus attend un évènement que seul un autre processus de l'ensemble peut provoquer.*

Exemple 17.3 *Supposons que deux processus veulent souhaitent enregistrer un document numérisé sur un scanner. Le processus A demande la permission pour utiliser le scanner et on lui accorde. Le processus B, codé différemment, demande lui en premier la permission au graveur CD, et on lui accorde. Les deux processus se sont alors bloqués l'un l'autre.*

Remarque 17.2

1. *Un interblocage peut arriver même avec un seul thread : s'il essaye de verrouiller un mutex qu'il a déjà verrouillé avant.*
2. *Un interblocage peut donc arriver même sans Entrées/Sorties avec le modèle précédent.*

B Ressources et conditions d'interblocages

Voici, sous sa forme résumée, la séquence d'évènement nécessaire pour utiliser une ressource :

— Sollicitation de la ressource.

— Utilisation de la ressource.

— Libération de la ressource

Théorème 17.1 (Conditions d'interblocage) *Pour qu'il y ait interblocage, les quatre conditions suivantes doivent être vérifiées :*

1. **Condition d'exclusion mutuelle** *: chaque ressource est soit attribuée à un seul processus, soit disponible,*
2. **Condition de détention et d'attente** *: les processus ayant déjà obtenu des ressources peuvent en demander de nouvelles.*
3. **Pas de réquisition** *: les ressources déjà détenues ne peuvent être retirées de force à un processus. Elles doivent être explicitement libérées par le processus qui la détient.*
4. **Condition d'attente circulaire** *: il doit y avoir un cycle d'au moins deux processus, chacun attendant une ressource détenue par un autre processus du cycle.*

Il y a quatre stratégies classiques pour traiter les interblocages :

— les ignorer ;

— les détecter et y remédier ;

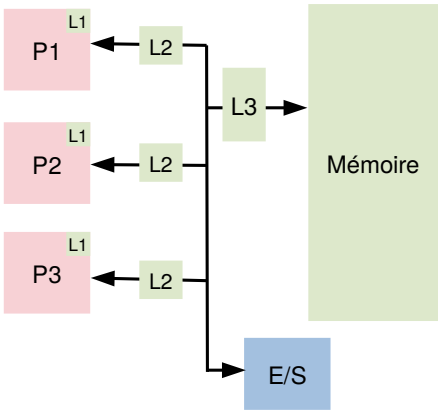
— les éviter de manière dynamique en allouant les ressources avec précautions (voir l'algorithme du banquier de Dijkstra) ;

— les prévenir en empêchant l'apparition d'une des quatre conditions de leur existence.

Remarque 17.3 *La première solution peut paraître bancale, mais elle reste possible si les interblocages sont peu fréquents et sans grande répercussion.*

En pratique, les méthodes comme l'algorithme du banquier de Dijkstra demandent beaucoup trop d'informations et sont trop contraignantes pour être implémenter. Il faut adopter des bonnes pratiques de conception et accepter le risque d'interblocage en général.

IV Intégration de la hiérarchie mémoire



L'intégration de la hiérarchie dans notre modèle posent de nouveaux problèmes de cohérences. Quand un processus écrit dans un cache L1 ou L2, si la même ligne est présente dans le cache d'un autre processus, il faudra mettre en place un mécanisme de synchronisation.

> **Remarque 17.4** *On s'intéresse ici à une vision horizontale de la cohérence des caches. Un autre problème de cohérence, lui verticale dans la hiérarchie mémoire, apparaît lorsqu'on écrit dans un cache.*

Le protocole MESI. Chaque ligne dispose d'un état qui est modifié par le contrôleur de cache selon la situation :

- *Modified* : la ligne a été modifié et n'est disponible que dans ce cache ;
- *Exclusive* : la ligne dans le cache est la même qu'en mémoire principale mais n'est que dans ce cache ;
- *Shared* : ligne non modifiée et partagée ;
- *Invalid* : la ligne a été invalidée, elle entraîne un cache miss.

Leçon 18

Stockage et manipulation de données, des fichiers aux bases de données.

Auteur·e·s: Émile Sorci
Niveau : L2
Pré-requis : Structures de données, C, bases de système
Références : [Dumas et al., 2007b], [Tanenbaum and Bos, 2014]

Motivations et problématiques liées au stockage Dès lors qu'il y a des données, des problèmes de stockage apparaissent :

1. **Stockage persistant et non volatile.** On souhaite sauvegarder des données sur le long terme et qui restent en mémoire après extinction de l'ordinateur.
2. **Efficacités.** Les accès au disque sont très fréquents ($\sim$ 10Go d'écritures par jour sur le disque pour un usage moyen d'un ordinateur) : ils doivent être efficaces.
3. **Données massives, stockage distribué et données partagées.**
4. **Questions de sécurité.**

I Les technologies majeures

Disque magnétique. Il est peu cher et moyennement rapide. Dû à son architecture et fonctionnement, les accès séquentiels sont rapides (à partir d'une position fixe, on lit les blocs qui suivent) mais les accès directs sont extrêmement lents à cause du mouvement de la tête (on part d'une position quelconque sur le disque pour accéder à une autre position donnée).

Disque SSD. Il est plus cher que les disques dures mais aussi beaucoup plus rapide. Il est constitué de circuits et n'a pas de partie mécanique. Les accès directs sont rapides et ont tous la même durée. Cependant les SSD ont un nombre d'écriture par cellule limité (typiquement d'une centaine à quelques milliers de To).

Remarque 18.1 *Le temps d'accès disque est très élevé par rapport au temps de calcul de l'UC.*

Les bandes magnétiques. Elles sont très peu chères mais très lentes. Elles servent uniquement à de l'archivage.

La technique RAID. On utilise plusieurs disques en parallèle avec un certain niveau de redondance (six niveaux). Cela permet la récupération de données en cas de panne d'un des disques.

II Les fichiers : une abstraction du stockage

Un **fichier** est une suite d'octets qui compose un ensemble de données.

Un **système de gestion de fichiers (SGF)** est un logiciel permettant de stocker les informations et de les organiser dans des fichiers sur des mémoires secondaires. Une telle gestion des fichiers permet de traiter et de conserver des quantités importantes de données ainsi que de les partager entre plusieurs programmes informatiques. Ce logiciel fait parti du système d'exploitation de l'ordinateur.

Il existe d'autres façons d'organiser les données, par exemple dans une base de données (notamment base de données relationnelle comme nous le verrons dans une partie suivante).

A Les attributs d'un fichier

Chaque fichiers possède plusieurs attributs tels que le nom (avec extension), la taille, la date de modification, les droits d'accès, etc...

B Organisation arborescente des répertoires

Un **répertoire** (ou un dossier) est une liste de descriptions de fichiers. Il est traité comme un fichier dont le contenu est la liste des fichiers référencés, et donc possède les mêmes attributs qu'un fichier classique (nom, taille, etc). Le répertoire dans lequel se trouve le navigateur de fichier est appelé **répertoire courant**.

Un **chemin d'accès** est une chaîne de caractère décrivant la position d'un fichier au sein du système de fichier. Il peut être relatif (au répertoire courant) ou absolu.

Exercice 18.1 (Commandes Shell Linux) *À quoi servent les commandes* `ls`, `cd`, `cat`, `touch`, `mkdir`, `rm`, `chmod` *et* `cp`. *À quoi correspondent les répertoires* `/`, `~` , `.` *et* `..`

C Manipulation des fichiers en C

Il existe plusieurs fonctions de manipulations des fichiers en C : `open`, `close`, `getc`, `fgets`, `fscanf`, `fprintf`.

Exercice 18.2 (TP) *Programmation d'un lexeur pour calculatrice à notation polonaise. Introduction à la sérialisation.*

D Autres outils pour la manipulation de fichiers

Compression. Il existe de nombreux algorithmes permettant de compresser un fichier : notamment le codage de Huffman et l'algorithme LZW. La commande `gzip` utilise un mélange des deux méthodes.
Développement 30. Présentation du codage de Huffman permettant de compresser des données, et preuve de son optimalité.

Encryption. Sans rentrer dans la théorie, on peut souvent avoir besoin d'encrypter un fichier. La commande gpg permet cela.

III Implémentation du système de fichier

Le disque est séparé en partitions :

MBR	Table des partitions	Partition 1	Partition 2	Partition 3	Partition 4

Chaque partition est elle-même divisée en plusieurs blocs :

- **Bloc d'amorçage** : initialise l'OS si cette partition en contient un.
- **Superbloc** : information sur le système de fichiers.
- **Gestion de l'espace libre** : structure de donnée pour gérer l'espace libres.
- **I-nodes** : voir ci-dessous.
- **Fichiers**

L'unité de base du disque est le **bloc** (typiquement 4kio).

A Allocation des blocs aux fichiers

Allocation contiguë. Si un fichier nécessite n blocs, on lui alloue n blocs consécutifs dans le disque. L'accès au fichier est efficace (peu de mouvement de la tête pour l'accès direct et séquentiel). Cependant, il entraîne un phénomène de fragmentation extrême et on doit connaître la taille du fichier dès le début de sa création.

Allocation par listes chaînées. Si un fichier nécessite n blocs, on lui alloue n blocs quelconques dans le disque et chaque bloc pointe sur le prochain. On résout le problème de fragmentation et on peut facilement modifier la taille d'un fichier. Il faut trouver une manière efficace de faire un accès direct à un fichier :

1. Un bloc par maillon, cette solution est peu efficace.
2. Table d'allocation des fichiers (FAT). Elle associe à un bloc l'indice du bloc suivant et est stockée en mémoire centrale. Cette solution est trop coûteuse pour de grands disques.
3. Les i-nodes : structure de donnée qui contient les blocs utilisés. Il s'agit d'un tableau d'indices des blocs. Il y a un nombre fini de blocs pour un fichier, donc une taille maximale. On ne garde en mémoire que les i-nodes des fichiers utilisés.

B Optimisation et amélioration des systèmes de fichiers

Plusieurs optimisations sont possibles.

— La taille du bloc doit être un bon compromis pour limiter la fragmentation mais garder une efficacité d'accès séquentiel.

— Lecture anticipée et cache en RAM.

— Utilisation d'un journal : on garde une trace des écritures en cours pour pouvoir les compléter dans le futur en cas d'arrêt brutal du système.

— Répartir les données sur le disque de façon à réduire le nombre de mouvements de la tête.

Remarque 18.2 *La plupart de ces optimisations ont été réalisées quand le disque magnétique était le seul moyen de stockage massif largement utilisé. Elles ne s'appliquent pas toutes aux SSD. En particulier, certaines sont même à éviter comme la défragmentation.*

IV Les bases de données

A SGBD et modèle relationnel

Définition 18.1 *Une* base de donnée *est un moyen de stocker des informations de manière structurée, de façon à pouvoir y accéder, les modifier, et les gérer, facilement à l'aide de requêtes plus ou moins complexes, souvent formulées dans un langage tel que le SQL.*

Les **systèmes de gestion de base de données** (SGBD) font partie des produits logiciels les plus vendus. En effet, les bases de données sont omniprésentes dans le monde moderne : ressources humaines, gestion de stock, de commandes en lignes, etc... Face à la multitude de situations dans lesquelles on utilise des BDD, et face à la multitude de manières de concevoir des bases répondant à un même problème, il est nécessaire de développer des méthodes systématiques permettant de concevoir des bases vérifiant certaines propriétés garantissant un bon fonctionnement.

Lien avec le système de gestion de fichier. Un système de fichiers est un logiciel qui gère et organise les fichiers sur un support de stockage, tandis que le SGBD est un logiciel utilisé pour accéder, créer et gérer des bases de données. Une différence majeure est qu'un SGBD fournit un mécanisme de récupération après incident, ce qui n'est pas le cas pour un système de fichier.

Modèle relationnel. Dans ce modèle, une base de données est un ensemble de **tables**, représentant des objets ainsi que leurs relations.

Exemple 18.1 *Voici un exemple de deux tables, ou relations, appelés* FILMS *et* SÉANCES.

FILMS

Titre	Réalisateur	Acteur
Titanic	*J. Cameron*	*L. Di Caprio*
Terminator	*J. Cameron*	*A. Schwarzenegger*
Inception	*C. Nolan*	*L. Di Caprio*
Une merveilleuse histoire du temps	*J. Marsh*	*E. Redmayne*

SÉANCES

Cinéma	Horaire	Titre
Le grand club	*18h*	*Titanic*
Le Palace	*18h30*	*Terminator*
Le Palace	*20h*	*Inception*
La Plage	*19h*	*Inception*

En pratique. On peut interagir avec une base de données, pour lire des données qui vérifient certaines conditions. Dans les SGBD, cela se fait à l'aide de requêtes dans un langage de programmation spécial, le plus répandu étant SQL.

Structure de donnée. En pratique, une base de donnée est stockée sur un disque externe, et les opérations les plus coûteuses en temps sont alors la lecture et l'écriture sur le disque. Ainsi que le SGBD optimise l'accès aux tuples d'une base de données, il est judicieux de recourir à une structure de donnée arborescente avec une ramification importante. C'est typiquement ce qu'offre la structure de **B-arbres**.
Développement 3. Présentation des B-arbres et complexité de l'algorithme de recherche.

B Clés primaires, clés étrangères

Clé primaire. On doit avoir un moyen de spécifier comment les tuples sont distingués au sein d'une même table. Les valeurs d'un tuple doivent permettre d'identifier uniquement chaque tuple.

Les clés primaires et étrangères sont deux concepts permettant d'imposer certaines contraintes, dites « d'intégrité », dans un schéma relationnel.

Définition 18.2 *Soit $R = \{t_1, ..., t_n\}$ une table ($t_1, ..., t_n$ sont les lignes).*

- *Une* **superclé** *est un sous ensemble d'attributs I tel que si $t_1, t_2 \in T$, alors $t_1.I \neq t_2.I$.*
- *Une* **clé primaire** *est une superclé minimale pour l'inclusion.*

Les clés primaires sont un moyen utile d'éviter les doublons : lors de l'insertion d'un nouvel enregistrement, un système de gestion peut vérifier que cela ne crée pas de doublon au niveau de la clé primaire de la table.

Clé étrangère. On peut ajouter une nouvelle contrainte à notre base de donnée. Si une de ses tables contient un certain attribut, on peut forcer ces valeurs à apparaître dans l'autre table.

Définition 18.3 *Soient R_1, R_2 deux tables sur les schémas $R_1[U_1]$ et $R_2[U_2]$, tels que R_1 possède parmi ses attributs une clé primaire pour R_2. Cet attribut ou ensemble d'attributs est appelé* **clé étrangère** *de T_1 référençant T_2.*

Les clés étrangères permettent de faire le lien entre différentes relations, et garantit l'intégrité référentielle du schéma. Cela signifie que lorsque l'on insère un enregistrement x dans R_1, on doit vérifier qu'il existe dans R_2 un enregistrement dont la valeur pour la clé primaire correspond à la valeur de x pour la clé étrangère .

Exemple 18.2 *Pour la table* FILMS*, l'attribut* **Titre** *est une clé primaire, et c'est une clé étrangère de* FILMS *référençant la table* SÉANCES*.*

C Introduction au langage SQL

Le langage SQL permet de communiquer avec une base de données.

La projection La *projection* d'une table en vue de l'obtention d'un ensemble de colonnes de cette table se fait via l'ordre SELECT.

Exemple 18.3 *Si on veut récupéré la table des titres et des noms des réalisateurs, on utilisera la requête suivante*

```
SELECT titre, réalisateur FROM Films
```

Restriction-Sélection. On peut faire des *restrictions* (ou *sélections*, mais le nom est trompeur) à l'aide du mot-clé WHERE. Là où les projections permettent de sélectionner des colonnes , les restrictions permettent de sélectionner des lignes de la table.

Exemple 18.4 *On cherche les séances ayant un prix inférieurs à 8€ :*

```
SELECT * FROM Séances
WHERE prix <= 8 ;
```

Jointures. Les **jointures** en SQL permettent d'associer plusieurs tables via une même requête, en associant certains attributs de chaque table.

Une jointure est donc un sous-ensemble du produit cartésien entre deux tables, obtenue via la requête `SELECT * FROM table1,table2,... ;`.

> **Remarque 18.3** *Dès qu'on combine plusieurs tables, on peut utiliser le nom des tables comme préfixe pour sélectionner un attribut spécifique. Par exemple, on pourra avoir une commande de la forme* `SELECT table1.id, table2.id FROM table1,table2 ;`.

On peut alors utiliser le mot-cl" `JOIN` pour resteindre ce produit cartésien selon une condition. La syntaxe est alors `SELECT ... FROM table1 JOIN table2 ON condition ;`.

Exemple 18.5 *On veut l'ensemble des séances avec le réalisateur et l'acteur principal en plus pour chaque films.*

```
SELECT *
FROM Séances
JOIN Films
ON Séances.titre = Films.titre ;
```

Remarque : *Cette commande peut aussi s'écrire de la manière suivante :*

```
SELECT *
FROM Séances, Films
WHERE Séances.titre = Films.titre ;
```

Leçon 19

Fonctions et circuits booléens en architecture des ordinateurs.

Auteur·e·s: Marin Malory
Niveau : L1
Pré-requis : Graphes, Éléments de logique mathématique
Références : [Patterson and Hennessy, 2017], [Perifel, 2014]

I Introduction

L'objectif de cette leçon est de donner poser les bases de la logique booléenne et des circuits booléens permettant de comprendre un cours d'architecture des ordinateurs.

Cette leçon permet notamment de faire des ponts avec d'autres domaines, notamment la théorie des graphes. Il s'agit aussi d'une application concrète de la logique booléenne vue une première fois en mathématiques.

Les blocs logiques sont catégorisés en deux types : ceux qui ne contiennent pas de mémoire et ceux qui en possèdent. Un bloc sans mémoire est dit **combinatoire** et la sortie d'un tel bloc dépend seulement des entrées. À l'inverse, dans un bloc avec mémoire, la sortie dépend de l'entrée et du contenu de la mémoire, appelé **état** du bloc logique. On parlera principalement ici de **logique combinatoire** et on abordera à la fin la **logique séquentiel**.

II Portes logiques et algèbre booléenne

A Tables de vérités et fonctions booléennes

Comme les blocs combinatoires ne contiennent pas de mémoire, ils peuvent être représentés par une fonction au sens mathématique.

Définition 19.1 *Une* **fonction booléenne** *à n entrées et m sorties est une fonction $f : \{0,1\}^n \to \{0,1\}^m$.*

Une représentation possible pour une telle fonction est une **table de vérité**. Étant donné une fonction booléenne à n entrées, sa table de vérités aura alors 2^n lignes.

Exemple 19.1 *On considère une fonction booléenne à trois entrées A, B et C et trois sorties D, E et F. On la définit comme suit : D vaut 1 si au moins une des trois entrées vaut 1, E vaut 1 si exactement deux entrées valent 1 et F vaut 1 si les trois entrées valent 1. La table contient $2^3 = 8$ lignes.*

A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

Proposition 19.1 *Il existe 2^{m2^n} fonctions booléenne à n entrées et m sorties.*

B Algèbre booléenne

Une autre approche pour définir une fonction logique est d'utiliser des **équations logiques**, via **l'algèbre booléenne**. En algèbre booléenne, toutes les variables sont soit 0, soit 1, et de manière classique, il y a trois opérations :

— l'opération OU, noté $+$ ou $\wedge$: le résultat de $A+B$ vaut 1 si et seulement si au moins une des deux variables vaut 1 ;

— l'opération ET, noté . ou $\vee$: le résultat de $A.B$ vaut 1 si et seulement si les deux variables valent 1 ;

— l'opération unaire NON, noté $\overline{\ldots}$ ou $\neg$: le résultat de $\overline{A}$ vaut 1 si et seulement si A vaut 0.

Il y a plusieurs lois qui permettent de manipuler ces équations :

— identité : $A+0=A$ et $0+A=A$;

— absorption : $A+1=1$ et $A.0=0$;

— inverse : $A+\overline{A}=1$ et $A.\overline{A}=0$;

— commutativité : $A+B=B+A$ et $A.B=B.A$;

— associativité : $A+(B+C)=(A+B)+C$ et $A.(B.C)=(A.B).C$;

— distributivité : $A.(B+C)=(A.B)+(A.C)$ et $A+(B.C)=(A+B).(A+C)$

Théorème 19.1 *Toute fonction booléenne $f:\{0,1\}^n\to\{0,1\}$ à n entrée et une sortie peut s'écrire comme une équation booléenne.*

Exemple 19.2 *La fonction $f:\{0,1\}^3\to\{0,1\}$ valant 1 si et seulement si exactement deux de ses entrées valent 1 peut se réécrire :*

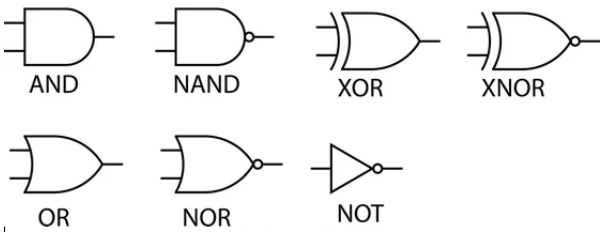
$$\begin{array}{rccl} f: & \{0,1\}^n & \to & \{0,1\} \\ & (A,B,C) & \mapsto & (A.B.\overline{C})+(A.\overline{B}.C)+(\overline{A}.B.C) \end{array}$$

Exercice 19.1 (Loi de De Morgan) *Montrer les lois de De Morgan.*

Remarque 19.1 *Il est possible d'étendre nos opérations en en rajoutant, comme par exemple l'opération OU EXCLUSIF (ou XOR).*

C Portes logiques

Définition 19.2 *Une porte logique est un dispositif (un composant électronique par exemple) qui implémente une fonction logique basique, comme le ET ou le OU.*



La plupart du temps, au lieu de dessiner explicitement un inverseur, on se contente d'ajouter une petite bulle sur une entrée ou une sortie.

Un **circuit logique** se présente comme un circuit électronique avec des entrées, des portes et une ou plusieurs sorties. Pour des soucis de stabilité, il ne doit pas y avoir de boucle.

Remarque 19.2 *Cette dernière condition nous dit que si on ignore la spécificité de chaque porte, le circuit est un fait un graphe orienté acyclique. Nous détaillerons cette formalisation dans la dernière partie.*

Exemple 19.3 (Un premier circuit logique) *On revient à notre exemple précédent, le circuit suivant implémente f.*

Exercice 19.2 (Implémentation de circuits logique simple)

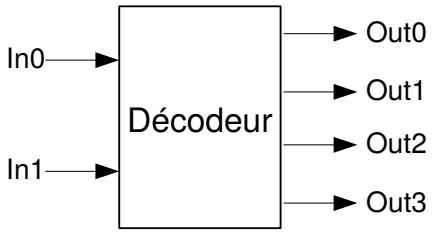
III Logique combinatoire

On s'intéresse ici à des circuits combinatoires très utilisés en architecture des ordinateurs.

A Décodeurs

Le premier bloc logique que nous allons construire est le **décodeur**. Le décodeur le plus commun possède n bits d'entrées et 2^n sorties : les n bits d'entrée peuvent être vu comme une adresse écrite en binaire, et la sortie vaut 1 pour cette adresse et 0 ailleurs.

Exemple 19.4 (Un décodeur 2 bits et sa table de vérité)



In0	*In1*	*Out0*	*Out1*	*Out2*	*Out3*
0	*0*	*1*	*0*	*0*	*0*
0	*1*	*0*	*1*	*0*	*0*
1	*0*	*0*	*0*	*1*	*0*
1	*1*	*0*	*0*	*0*	*1*

Exercice 19.3 *Donner la table de vérité d'un décodeur 3 bits.*

B Multiplexeurs

Une fonction booléenne souvent utilisé est le **multiplexeur**, aussi appelé un **sélecteur**. Regardons le multiplexeur à 2 bits ci-dessous : ils possèdent 3 entrées, deux valeurs et un **bit de contrôle**. Ce dernier bit permet de choisir l'entrée qui choisit comme sortie.

De manière générale, un multiplexeur permet d'aiguiller une entrée parmi 2^n vers un bit de sortie à l'aide de n bits de contrôles (eux aussi en entrée).

De manière général, un multiplexeur peut être découper en trois parties :

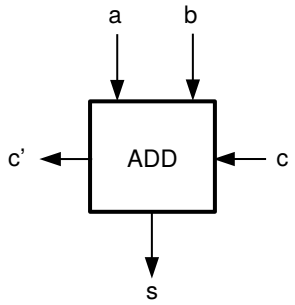
1. un décodeur qui génère 2^n signaux à partir des n bits de contrôle, chacun indiquant une valeur différente d'entrée ;
2. un tableau de 2^n portes ET, chacun combinant une des entrées avec un bit venant du décodeur ;
3. une grande porte OU qui combine les sorties des portes ET.

Exercice 19.4 *Dessiner un multiplexeur 8 vers 1.*

C Additionneur 1-bit et full-adder

Le principal intérêt des circuits logiques est la possibilité de réaliser des opérations logiques et arithmétiques sur des vecteurs de bits. On commence ici par l'opération la plus simple : l'addition.

Voici l'interface d'un additionneur 1 bit, qui additionne deux bits a et b avec une retenue entrante c, et renvoie un bit de sortie ainsi qu'une retenue sortante c'.



Exercice 19.5 *Donner la table de vérité de l'additionneur un bit, ainsi qu'un circuit qui l'implémente.*

On peut alors construire un additionneur n bits en juxtaposant n additionneurs un bit. Le circuit obtenu est appelé **full-adder**.

Le principal problème de ce circuit est le temps de propagation de la retenue. Ce temps est proportionnel au **chemin critique** du circuit : c'est le plus long chemin dans le circuit. Dans le cas du full-adder, ce chemin est de longueur $\mathcal{O}(n)$.

Additionneur rapide. Il est possible de construire un additionneur plus rapide en propageant la retenue de manière astucieuse.
Développement 22. Construction d'un additionneur rapide.

D Read Only Memory (ROM)

Les circuits combinatoires peuvent aussi permettre de construire des mémoires à lecture seule. Une ROM est appelée mémoire parce que certains endroits peuvent être lus, mais ces endroits sont fixés (le plus souvent à la fabrication). Ainsi, une ROM reçoit en entrée une adresse et retourne une certaine sortie selon l'adresse reçue.

> **Remarque 19.3** *Il s'agit ni plus ni moins que l'implémentation d'une fonction logique classique. La seule différence est conceptuelle, puisque ici l'entrée est en fait une adresse.*

IV Formalisation

On présente ici une formalisation de la notion de circuit [Perifel, 2014], permettant de travailler plus théoriquement sur ceux-ci via la théorie des graphes. Ces définitions trouvent aussi leur place en théorie de la complexité, où les circuits booléennes forment un modèle de calcul équivalent aux machines de Turing.

A Définitions

L'idée consiste à voir un circuit comme un graphe orienté qui, pour des raisons de stabilité, ne contient pas de circuit.

Définition 19.3 (Circuit booléen) *Un circuit booléen est un graphe orienté acyclique (et connexe) tel que :*

- *les sommets de degré entrant nul sont appelés* **entrées** *et sont étiquetés par le nom d'une variable x_i, ou par la constante 0 ou 1 ;*
- *les sommets de degré entrant 1 sont appelés* **portes de négations** *et sont étiquetés par $\neg$;*
- *kes autres sommets sont soit des* **portes de conjonction**, *soit des* **portes de disjonction**, *respectivement étiquetés par $\wedge$ et $\vee$;*
- *les sommets de degré sortant nul sont appelés* **sorties** *;*
- *les voisins entrants d'une porte sont appelés* **arguments**.

Sauf mention du contraire, nos circuits auront une sortie **unique** *et les portes $\vee$ et $\wedge$ auront un degré entrant égal à 2.*

Enfin si C est un circuit, on appellera **sous-circuit** *de C issu d'une porte γ le circuit dont la sortie est γ et dont les sommets sont tous les prédécesseurs de γ dans G.*

Exemple 19.5 *Exemple simple.*

Un circuit booléen calcule une fonction booléenne, définie de manière naturelle comme suit :

Définition 19.4 *Soit C un circuit booléen sur les variables $x_1, ..., x_n$. La fonction calculée par une porte γ de C est la fonction $f_\gamma : \{0,1\}^n \to \{0,1\}$ définie récursivement comme suit :*

- *si γ est une entrée étiqueté par une variable x_i, alors $f_\gamma(x_1, ..., x_n) = x_i$;*
- *si γ est une entrée étiquetée par une constante $c \in \{0,1\}$, alors $f_\gamma(x_1, ..., x_n) = c$;*
- *si γ est une porte $\neg$ dont l'argument est la porte γ', alors $f_\gamma = \neg f_\gamma$;*
- *si γ est une porte $\wedge$ (resp. $\vee$) dont les arguments sont les portes $\gamma_1, ..., \gamma_k$, alors $f_\gamma = \bigwedge_{i=1}^k f_{\gamma_i}$ (resp. $f_\gamma = \bigvee_{i=1}^k f_{\gamma_i}$)*

Si le circuit C a k portes de sorties $s_1, ..., s_m$, la fonction calculée par C est alors la fonction $f : \{0,1\}^n \to \{0,1\}^m$ dont la ième composante est la fonction f_{s_i}.

B Mesures

On peut maintenant définir deux caractéristiques importantes des circuits :

Définition 19.5 *Soit C un circuit booléen.*

- *La* **taille** *de C, noté $|C|$, est le nombre de sommets du graphe.*
- *La* **profondeur** *de C est la taille maximale d'un chemin d'une entrée à une sortie.*

On peut faire un lien entre la taille et la profondeur.

Lemme 19.1 *Soit C un circuit de profondeur p ayant une unique sortie. Si les portes ont un degré entrant majoré par k (pour un entier $k \geq 2$), alors $p < |C| \leq k^{p+1} - 1$.*

On peut aussi majorer le nombre de circuits d'une certaine taille.

Lemme 19.2 *Soit t et k des entiers. Si les portes ont un degré entrant majoré par k, alors il y a au plus $3^t t^{(k+1)t}$ circuits de taille $\leq t$.*

Exercice 19.6 (Circuits inverseurs) *On appelle* **circuit n-inverseur** *un circuit ayant n entrées booléennes $b_1, ...b_n$ (et éventuellement deux entrées `V` et `F`, et dont les n sorties sont $\neg b_1$,...., $\neg b_n$.*

1. *Montrer qu'un circuit 1-inverseur contient au moins une porte `NON`.*
2. *Montrer qu'un circuit 2 ou 3-inverseur contient au moins deux portes `NON`.*
3. *Montrer qu'un circuit n-inverseur contient au moins $\lfloor \log_2(n) \rfloor + 1$ portes `NON`.*

Calcul de la profondeur. Le calcul de la profondeur d'un graphe orienté acyclique nous permet de calculer la longueur du chemin critique d'un circuit booléen. Ce calcul est possible en temps linéaire en la taille du graphe.
Développement 23. Calcul de la profondeur d'un DAG.

Leçon 20

Principes de fonctionnement des ordinateurs : architecture, notions d'assembleur.

Auteur·e·s: Sorci Emile
Niveau : L2 à L3
Pré-requis : Notions d'encodage de l'information, logique booléenne, circuits booléens, notions de base de programmation (langage C)
Références : [Patterson and Hennessy, 2017], [Tanenbaum, 1976], [Stallings, 2003]

I Introduction

A Motivation du cours

Ce cours permet de comprendre le fonctionnement de l'objet ubiquitaire qu'est l'ordinateur. En plus de la compréhension de l'informatique de bas niveau (proche de la machine), ce cours permet notamment de motiver certains aspects plus théoriques de l'informatique.

B Historique et mise en contexte

Historique Même si l'informatique a explosé au XXème siècle, cette science a débuté avant. Une de premières machine programmable était une machine à tisser perforée. Plus tard, **Ada Lovelace** est à l'origine du premier programme informatique, codé sur la machine analytique de **Charles Babbage**, que l'on peut voir comme l'ancêtre de l'ordinateur. Au XXème siècle, **Turing** et **Von Neumann** ont chacun proposés des modèles de machines programmables et universels, menant aux premiers ordinateurs universels (voir ENIAC). Les avancées technologiques ont menés aux ordinateurs actuels, avec notamment quatre étapes technologiques : les tubes à vide, les transistors, les circuits intégrés et la Very Large Scale Integration.

Explosion des performances Depuis les premiers ordinateurs, on a pu observer une augmentation exponentielle des puissances de calcul, de la taille des mémoire, etc. La **loi de Moore** (1965) affirme que les ressources des circuits intégrés doublent tous les 18-24 mois.

> **Remarque 20.1** *Actuellement, la loi de Moore a atteint ses limites physiques. On tente de la conserver par l'ajout du parallélisme qui permet d'augmenter les performances.*

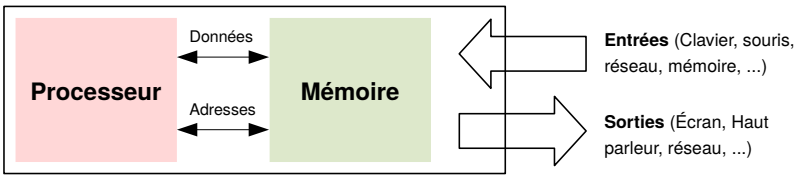
Où on en est actuellement De nos jours, les ordinateurs sont omniprésents (explosion du nombre d'ordinateurs existants) : pc, tablettes, smartphones, ordinateurs de bord dans les voitures, objets connectés (iot), serveurs et super calculateurs. Par exemple, plusieurs milliards d'objets sont connectés à Internet.

Aspects sociétaux De grands enjeux politiques et économiques se jouent dans l'industrie et la recherche de l'informatique. L'écologie est aussi à prendre en compte, autant dans la conception (on parle d'éco-conception) que dans la prédiction de la consommation (effet rebond).

II Grands principes de fonctionnement des ordinateurs de nos jours

A Architecture de von Neumann

Une **mémoire** (unité de stockage) qui contient les programmes et les données. Un **processeur** (unité de calcul) qui effectue des actions selon les instructions qu'il lit en mémoire. Avec ces deux principaux éléments, on peut donner le modèle de l'**architecture de Von Neumann** :



La mémoire est **adressable**. C'est-à-dire qu'elle est séparée en cases qui contiennent un nombre fixe de bits (un octet typiquement). Et chaque case a une adresse.

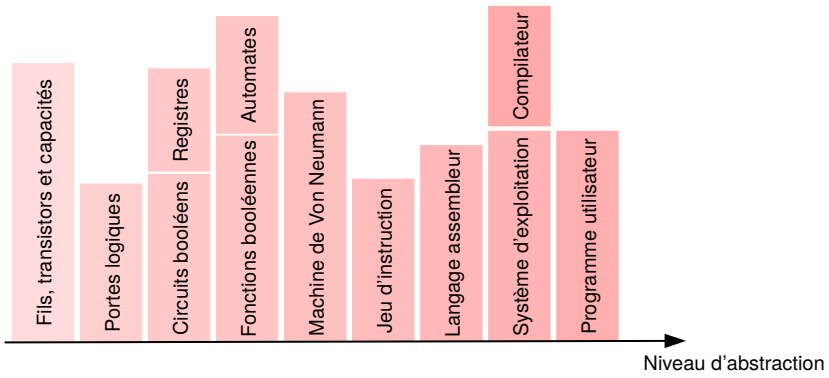
Le processeur accède à la mémoire par blocs de 8, 16, 32 ou 64 bits appelés **mots mémoire** (de nos jours, les architectures 64 bits sont les plus courantes). Pour cet accès, le processeur utilise un registre appelé **compteur ordinal**(*Program Counter* ou PC) qui contient une adresse en mémoire.

Cycle de von Neumann :

1. Lire le mot mémoire qui commence à l'adresse PC ;
2. Interpréter ce mot mémoire comme une instruction et l'exécuter ;
3. Augmenter le PC pour passer à l'instruction suivant et recommencer.

B Utiliser l'abstraction pour simplifier un système hiérarchique

Une notion importante en architecture des ordinateurs est l'**abstraction**. De la même manière qu'on préfère écrire un programme en Python plutôt qu'en binaire, plusieurs niveaux d'abstractions se distinguent dans un ordinateur.



III Jeu d'instruction et assembleur

A Définitions

Une **instruction machine** est une séquence de bits que le processeur peut exécuter. Un **jeu d'instruction (ISA)** pour *Instruction Set Architecture* est une abstraction qui fait l'interface entre le matériel

et le logiciel. Il définit entre autres l'ensemble des instructions valides, ce qu'elles font, leur donne une **mnémonique** (une représentation textuelle qui facilite l'utilisation pour les humains), le type des données supportées, le nombre de registres, leur utilisation ainsi que les drapeaux.

Le **langage assembleur** est un langage de programmation défini par l'ensemble des mnémoniques étendu notamment pour permettre l'écriture de fonctions.

Exemple 20.1 *On observe certains de ces éléments sur le jeu d'instruction RISC-V 32 bits (RV32I). On montre notamment les instructions simples comme* `add rs1 rs2 rd`, `and rs1 rs2 rd` *et* `lw rd rs1(offset)`.

Deux grandes catégories d'ISA :

CISC : (Complex Instruction Set Computer) se dit d'un jeu d'instruction avec de nombreuses et longues instructions spécifiques (exemple : processeur intel x86). Ce paradigme qui met en avant la puissance de calcul d'une instruction.

RISC : (Reduces Instruction Set Computer) se dit d'un jeu d'instruction avec peu d'instructions courtes, souvent de même longueur (exemple : les processeurs ARM et RISC-V). Ce paradigme met en avant la simplicité d'usage et d'implémentation.

B Spécification d'un jeu d'instruction RISC-V réduit sur 32 bits

En partant d'un exemple de programme écrit en C, on observe nos besoins pour construire des instructions simples. On se rend compte que l'on a besoin de plusieurs types d'opérations :

— Opérations arithmétiques et logiques ;

— Opérations de transferts de données avec la mémoire ;

— Opérations de branchement conditionnel et inconditionnel.

L'ISA spécifie aussi qu'il y a 32 registres de 32 bits en plus de PC,nommés `x0, ..., x31`, que `x0` est toujours à 0, que l'espace d'adressage est de 2^{30}, des mots de 32 bits dont le bit de poids faible est 0.

> **Remarque 20.2** *L'ISA contient d'autres informations que nous ignorons par simplicité et pour ne pas alourdir le propos.*

Chacun des types d'instructions a un format de code machine spécifique, séparé en champs. L'un d'eux est l'**opcode** qui permet de retrouver le type d'instruction.

Exercice 20.1 *Quelques exercices :*

1. *À l'aide d'une spécification détaillée de l'ISA RV32I, effectuer quelques traductions entre assembleur et langage machine.*
2. *Détailler l'exécution (contenu des registres et PC) pour un programme simple écrit en assembleur.*
3. *"Traduction" d'un programme simple écrit en C vers l'assembleur : initiation à la compilation.*

IV Implémentation physique d'un jeu d'instruction

A Synchronicité

Le **delai d'une porte logique** est le temps entre l'arrivée d'un signal stable et la stabilisation du signal en sortie. Le **délai d'un circuit combinatoire** est la plus grande somme de délais entre une entrée et une sortie.

Lien avec la théorie des graphes. Un circuit booléen peut être vu comme un graphe orienté acyclique. Le délai su circuit est alors lié au chemin critique dans ce graphe.
Développement 23. Recherche de chemin critique dans un circuit booléen.

Il faut faire attention au moment de la mise à jour du contenu des registres.

Exemple 20.2 *Donner un cas où le circuit ne fonctionne pas comme prévu à cause de problèmes de synchronicité.*

On **synchronise** alors les registres à l'aide d'une horloge (signla en créneau de période fixe en général) commune qui arrive à tous les registres de façon synchrone.

Exemple 20.3 *Détail des états d'un circuit en utilisant un chronogramme.*

Remarque 20.3 *On évite de mettre des portes logiques sur le signal d'horloge.*

B Automates comme abstraction des circuits séquentiels

Un **circuit séquentiel** est un circuit de portes logiques contenant des registres.

Exemple 20.4 *Circuit d'un compteur*

Une **machine de Moore** est une abstraction des circuits séquentiels. Elle est définie par un ensemble d'entrée E, de sortie S, des états Q (abstraction de l'état des registres), une fonction de transition T : $Q \times E -> Q$ (abstraction d'un circuit booléen) et d'une fonction de sortie $F : Q -> S$ (abstraction d'un autre circuit booléen). On dit que c'est un automate.

Exemple 20.5 *Construction l'automate pour le circuit du compteur.*

C Implémentation d'un processeur RISC-V simplifié

Un processeur est composé de deux parties :

— Le **chemin de données** : circuit qui effectue les calculs, c'est les muscles de la machine ;

— L'**unité de contrôle** : circuit qui active et désactive certaines parties du chemin de données, c'est le cerveau.

Exemple 20.6

— *Implémentation d'une ALU simplifiée.*

— *Implémentation du chemin de données et description de l'automate de contrôle.*

V Amélioration de la performance grâce au parallélisme

A Différents niveaux de parallélisme et classification de Flynn

— Parallélisme au niveau des bits : additionneurs rapides

— Parallélisme au niveau des données : instruction vectorielle

— Parallélisme d'instruction : VLIW et pipeline

— Parallélisme de tâche : Utilisation de threads et plusieurs processus

Développement 22. Construction d'un additionneur rapide *Carry Look-Ahead*.

Classification de Flynn : SISD (processeur RV32I), SIMD (processeurs vectoriels), MIMD (ordinateur multi-processeur) et MISD (n'existe pas).

B Approfondissement du fonctionnement d'un pipeline

Le **pipelining** est une techinque d'implémentation dans laquelle de multiples instructions sont superposées en exécution pour augmenter le **débit** de traitement des instructions par processeur.

Exemple 20.7 *Un exemple profane : on fait des cookies, on peut préparer la pâte pendant que la fournée précédente cuit et que celle d'avent refroidit. On introduit le* **diagramme d'exécution**.

Proposition 20.1 *Si on découpe le traitement d'une instruction en k étapes, on peut alors espérer que pour un processeur pipeliné, le temps d'exécution pour un grand nombre d'instructions soit divisé par k.*

Dans le cas du RISC-V, il y a en général 5 étapes identifiables : chargement de l'instruction, décodage de l'instruction, lecture des registres, exécution et écriture dans un registre.

Remarque 20.4 *En pratique, du aux dépendances, il faut parfois faire une pause dans le pipeline. On appelle cela* **buller**.

Exemple 20.8 *Donner le diagramme d'exécution d'un programme assembleur avec dépendances de données et de contrôle. On pourra aborder la prédiction de branchement.*

Leçon 21

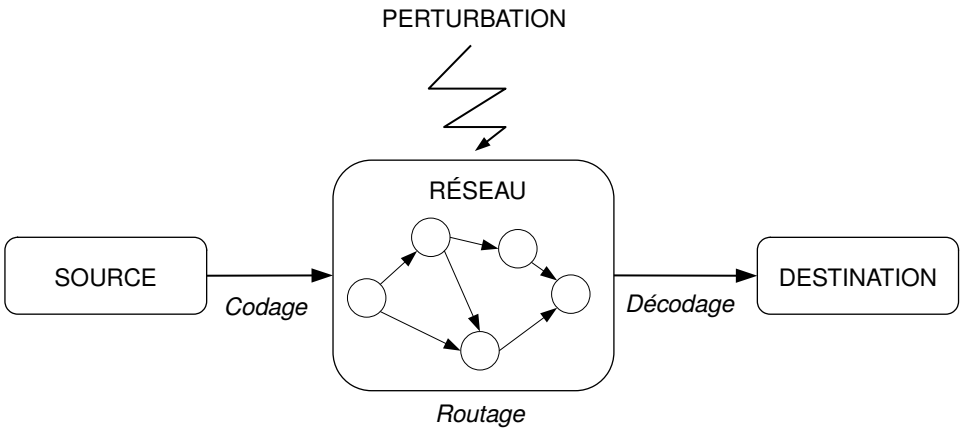
Échanges de données et routage. Exemples.

Auteur·e·s: Marin Malory
Niveau : L1
Pré-requis : algorithmique de graphe, notions de complexité
Références : [Dumas et al., 2007b], [Cormen et al., 2009], [Benoit et al., 2013], [Tanenbaum and Wetherall, 2011]

I Introduction

Cette leçon a pour but de traiter la transmission automatique d'informations numériques. La transmission de ce type de données est omniprésente dans la technologie, et spécialement dans les télécommunications. Il est donc nécessaire de s'appuyer sur de bonnes bases théoriques pour que cette transmission soit fiable, ce dernier terme se voyant donner plusieurs significations.

En particulier, cette leçon se focalisera sur deux aspects de la transmission de données : le codage et le routage.



La communication d'une information commence par sa formulation par un émetteur, se poursuit par un transit par un canal et se termine par la reconstitution du message par le destinataire. L'information doit être reçue par son destinataire dans son intégralité, en sécurité, et le plus rapidement possible. La première partie s'intéressera à la formulation et la reconstitution d'un message, tandis que la seconde à la transmission.

II Codage

On s'intéresse ici à la formulation d'une information par l'émetteur ainsi qu'à sa reconstitution par le destinataire. On s'intéresse à trois problématiques majeures :

— l'**efficacité de la transmission** : compression des données ;

— la **sécurité de l'information** : cryptage, authentification ;

— l'**intégrité du message** : correction d'erreurs.

Pour formaliser les messages sources et les codes, on définit le langage dans lequel ils sont exprimés. On appelle **alphabet** un ensemble fini Σ d'éléments appelés **caractères**. Une suite de caractère de Σ est appelée **chaîne** ou **mot**, et on note Σ^* l'ensemble des chaînes de Σ. Comme alphabet du code et l'alphabet de la source peuvent différer, on parle d'**alphabet source** et d'**alphabet de code**.

Exemple 21.1 (Image) *Supposons qu'un scanner nous retourne une suite de pixel et qu'on veuille les coder en suite de bits, l'alphabet source est* $\{\square, \blacksquare\}$ *et l'alphabet de code est* $\{0, 1\}$.

A Compression du message

Un premier exemple de compression. On reprend l'exemple de notre image. Par exemple, dans de nombreux cas (pour le scan d'une feuille blanche par exemple), il arrive très souvent que les bords soient blancs et que le message par exemple 1025 pixels blancs. Au lieu de transmettre directement le message comprenant 1025 fois 0, on peut transmettre $[1025]_2 0$, ce qui s'écrit en binaire $10000000001\ 0$.

Codage de Huffman. De manière plus formelle, on peut définir un code comme une fonction. En effet, un code binaire est une fonction injective $c : \Sigma \to \{0,1\}^*$, qui peut être étendue en une fonction $c : \Sigma^* \to \{0,1\}^*$ par concaténation. On présente ici le **codage de Huffman**, qui repose notamment sur l'utilisation d'arbres binaires.
Développement 30. Présentation du codage de Huffman permettant de compresser des données, et preuve de son optimalité.

À partir de maintenant, le message transmis est un donc une suite de bits représentant une information.

B Détection d'erreur

La fiabilité d'un canal est loin d'être parfaite. Par exemple, les canaux téléphoniques ont un taux d'erreur variant de 10^{-4} à 10^{-7}, ce qui est loin d'être négligeable pour des longs messages. Il existe des moyens de détecter et même de corriger des erreurs de transmissions lorsqu'on reçoit un message. Ils sont tous basés sur le même principe : ajouter de l'information pour vérifier la cohérence du message reçu.

Un **codage par blocs** de taille k consiste à découper le message $m \in \{0,1\}^*$ en blocs M_i de k symboles, en traitant chaque bloc l'un après l'autre.

Afin de se protéger contre des erreurs en les détectant voire en les corrigeant de manière automatique, les méthodes de codage par blocs ajoutent de la redondance aux k symboles d'information du bloc initial.

Définition 21.1 *Un code* (n, k) $(n > k)$ *est un ensemble* $C_\Phi = \{\Phi(s), s \in V^k\}$ *où* $\Phi : V^k \to V^n$.
Le **rendement** R *d'un code* (n, k) *est le taux* $R = \frac{k}{n}$.

Lorsqu'on reçoit au bout du canal un mot de code qui n'est pas dans C_Φ, il y a eu une erreur. Il y a deux cas :

— on corrige directement le mot via les bits supplémentaires ;

— le récepteur demande de ré-envoyer le message.

Un exemple simple de détection par parité. On code le mot $m = s_1...s_k$ par $\Phi(m) = s_1...s_k c_{k+1}$ où $c_{k+1} = (\sum_{i=1}^{k} s_i) \mod 2$. Autrement dit, $c_{k+1} = 0$ si et seulement si m contient un nombre pair de 1. On peut alors faire un **contrôle de parité** qui permet de détecter une erreur si un nombre impair de bits ont changé.

Un exemple simple de correction directe par parité longitudinale et transversale. On construit un code (n, k_1k_2) binaire comme suit. Un message source m peut être représenté comme une matrice à k_1 lignes et k_2 colonnes. Le mot de code $\Phi(m)$ associé contient $(k_1 + 1)(k_2 + 1)$ bits dont $k_1 + k_2 + 1$ bits de parités. Avec ce code, on peut détecter et supprimer une erreur portant sur un bit en localisant la ligne et la colonne.

C Le chiffrement

Supposons maintenant, avant d'avoir compressé le message et apporté un format qui permette la détection d'erreurs, que nous voulions garder le message secret pour toute personne excepté son destinataire. Le canal téléphonique, comme la plupart des canaux, ne permet pas de garder le secret en lui-même. Tout message qui y transite peut être facilement lu par un tiers.

Codage de César. Pour illustrer le codage de César, on suppose que l'alphabet source est l'alphabet latin, et on numérote les lettres de 0 à 25. En supposant que l'émetteur et le destinataire ont choisi discrètement un entier K, appelé **clé**, alors chaque lettre x est codé par $f_K(x) = x + K \mod 25$. De manière plus générale, si l'alphabet source contient n caractères, un **codage de César** est une fonction $f_K : x \mapsto x + K \mod n$.

Exemple 21.2 *Pour l'alphabet latin et $K + 3$, le mot INFORMATIQUE est codé en LQIRUPDWLTXH.*

D'autres systèmes cryptographiques plus élaborées existent, comme le chiffrement affine ($f_{a,b}(x) = a.x + b \mod n$) ou encore les chiffrements par substitution où chaque lettre par un symbole d'un autre alphabet.

D Décodage

On présente maintenant l'opération inverse du codage : le décodage. Le message arrive, on peut vérifier s'il contient des erreurs et éventuellement les corriger et ensuite le décompresser. Il se peut alors que le message soit crypté.

Déchiffrement. Si on est bien le destinataire, alors on connaît un moyen pour retrouver le message initial, on parle de **déchiffrement**. Dans le cas du codage de César, pour décoder un message y, il suffit de prendre $y - K \mod n$.

L'attaque. Pour qui ne possède pas la clef ou la méthode de chiffrement, on parle d'**attaque** sur le code, ou de **cassage** du code. La discipline qui consiste à inventer des méthodes d'attaque pour briser les codes existants ou pour construire des codes plus résistants est la **cryptanalyse**. Pour le codage de César, un algorithme d'attaque consiste à tester l'ensemble des décalages possibles.

Exercice 21.1 *Donner un algorithme permettant de casser un chiffrement affine. Donner sa complexité.*

> **Remarque 21.1** *Dans certains cas, il se peut que lors de la phase de décompression, il y ait des pertes de données. On utilise souvent des codes qui permettent un certain niveau de perte, quand on estime que l'information importante n'est pas altérée. C'est souvent le cas pour les informations audiovisuelles.*

III Routage

En pratique, un canal ne va pas directement d'un point A à un point B, mais il fait partie d'un réseau. De manière théorique, un réseau est un graphe pondéré $G = (V, E, w)$ où les sommets sont soit des **hôtes** (émetteurs ou destinataires) ou des intermédiaires appelés **routeurs** et les arêtes symbolisent une communication possible entre les sommets. Un **algorithme de routage** est un algorithme de décision à l'origine du choix du chemin que vont emprunter les données dans un réseau.

Pour assurer l'efficacité, on cherche donc à transmettre les données de manière rapide. Cela revient la plupart du temps à trouver un plus court chemin.

IV Plus court chemin dans un graphe pondéré

Définition 21.2 (Graphe pondéré) *Un graphe pondéré est un graphe $G = (V, E)$ muni d'une fonction de poids $w : E \to \mathbb{R}$. On peut étendre w à V^2 en posant $w(u, v) = +\infty$ lorsque $(u, v) \notin E$.*
Le poids d'un chemin est alors défini comme la somme des poids des arêtes qui le compose.

Définition 21.3 (Distance dans un graphe pondéré) *Étant donné un graphe pondéré (G, w) dans circuit de poids strictement négatif, on définit la distance de u à v par :*

$$d(u, v) = \min\{w(c) | c \text{ est un chemin reliant } u \text{ à } v\}$$

A Algorithme de Floyd-Warshall

On essaye ici de trouver les plus courts chemins pour toutes les paires de sommets. On supposera $G = (V, E, w)$ avec $V = \{0, ..., n-1\}$, et si $ij \notin E$, alors on pose $w(ij) = +\infty$.

Cet algorithme utilise la programmation dynamique : en notant $d_{ij}^{(k)}$ la distance du plus court chemin allant de i à j passant par les k premiers sommets, on a

$$\begin{cases} d_{ij}^{(k)} = w_{ij} & \text{si } k = 0 \\ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{si } k \geq 1 \end{cases}$$

L'algorithme de Floyd-Warshall implémente directement cette idée avec une complexité temporelle $\mathcal{O}(|V|^3)$.

B Algorithme de Bellman-Ford

On essaye ici, étant donné une source s, de trouver l'ensemble des plus courts chemin de s aux autres sommets.

Théorème 21.1 *L'algorithme de Bellman-Ford est correct et s'exécute en temps $\mathcal{O}(n \times m)$ où n est le nombre de sommets et m le nombre d'arcs.*

C Nombre maximums de chemins disjoints

Jusqu'à maintenant, les algorithmes proposent juste le plus courts chemin pour chaque requête. Cependant, lorsque plusieurs requêtes arrivent en même temps, on peut vouloir que les chemins empruntés soient disjoints pour des raisons de fiabilité. Le problème d'optimisation sous-jacent est alors bien plus complexe.

Cette partie, qui mène ensuite à un développement, n'est pas abordable à niveau L1 mais pourra être abordé plus tard dans le cursus des étudiants afin de mettre en lien certaines méthodes théoriques avec des problématiques apparaissant en routage.

Algorithme 21.1 : Bellman-Ford(G,s)

Données : Un graphe orienté pondéré $G = (V, E, w)$ et une source $s \in V$
Résultat : Plus courts chemin de s aux autres sommets.
pour $u \in V$ **faire**
    distance$[v] \leftarrow +\infty$;
    pred$[v] \leftarrow$ NIL ;
distance$[s] \leftarrow 0$;
pour $i = 1...|V| - 1$ **faire**
    pour $(u, v) \in E$ **faire**
        si *distance*$[v] >$ *distance*$[u] + w(u, v)$ **alors**
            distance$[v] \leftarrow$ distance$[v] + w(u, v)$;
            pred$[v] \leftarrow u$;
retourner *distance, pred*

Définition 21.4 (Maximum-Edge Disjoint Path (MEDP))

— **Données** *: un graphe $G = (V, E)$ et un ensemble de requêtes $R \subset V \times V$.*

— **Sortie** *: nombre maximal de requêtes réalisables, en sachant que cet ensemble est réalisable s'il existe un ensemble de chemins deux à deux à arêtes disjointes qui réalise les requêtes.*

Ce problème s'avère être $\mathbf{NP}$-complet : on ne connaît pas d'algorithme permettant de le résoudre en temps polynomial, mais on peut trouver des approximations.

Théorème 21.2 *Il existe une $(n-1)$-approximation pour le problème $\mathbf{MEDP}$, où n est le nombre de sommets du graphe en entrée.*

Théorème 21.3 *Il existe une $\lceil \sqrt{m} \rceil$-approximation pour le problème $\mathbf{MEDP}$, où m est le nombre d'arêtes du graphe en entrée.*

Développement 9. Étude d'un algorithme d'approximation pour le problème $\mathbf{MEDP}$.

D Algorithmes distribués

Les algorithmes précédents nécessitent de connaître la topologie du graphe en entier. Ce n'est pas toujours le cas, et cette topologie peut être variable (en cas de panne d'un routeur par exemple). En pratique, les algorithmes utilisés sont donc distribués.

Routage par vecteur de distance. Pour cet algorithme, chaque routeur maintient une table lui donnant les distances les plus courtes pour chaque destination, et aussi le prochain lien à suivre pour chaque chemin. Les tables sont mis à jour en échangeant des informations avec ses voisins. Cet algorithme est aussi appelé l'algorithme de Bellman-Ford distribué. Il s'agissait de l'algorithme de routage de ARPANET, et est aussi utilisé par l'Internet sous le nom de RIP.

Routage par état fini. Dans cet algorithme, chaque routeur doit :

1. Découvrir ses voisins et connaître leurs adresses ;
2. Initialiser les distances (ou coût) avec ses voisins ;
3. Construire un paquet contenant tout ce qu'il vient d'apprendre ;

4. Envoyer son paquet et recevoir celui des autres ;
5. Calculer le plus court chemin vers tous les autres routeurs (via l'algorithme de Dijkstra).

Leçon 22

Modèle relationnel et conception de bases de données.

Auteur·e·s: Rousseau Guillaume, Marin Malory
Niveau : L3
Pré-requis : Logique du 1er ordre, SQL
Références : [Silberschatz et al., 2020], [Pinchinat et al., 2022]

Introduction

Définition 22.1 *Une* base de donnée *est un moyen de stocker des informations de manière structurée, de façon à pouvoir y accéder, les modifier, et les gérer, facilement à l'aide de requêtes plus ou moins complexes, souvent formulées dans un langage tel que le SQL.*

Les **systèmes de gestion de base de données** (SGBD) font partie des produits logiciels les plus vendus. En effet, les bases de données sont omniprésentes dans le monde moderne : ressources humaines, gestion de stock, de commandes en lignes, etc... Face à la multitude de situations dans lesquelles on utilise des BDD, et face à la multitude de manières de concevoir des bases répondant à un même problème, il est nécessaire de développer des méthodes systématiques permettant de concevoir des bases vérifiant certaines propriétés garantissant un bon fonctionnement.

I Modèle relationnel

Exemple 22.1 *Voici un exemple de deux tables, ou relations, appelés* FILMS *et* SÉANCES.

FILMS

Titre	Réalisateur	Acteur
Titanic	*J. Cameron*	*L. Di Caprio*
Terminator	*J. Cameron*	*A. Schwarzenegger*
Inception	*C. Nolan*	*L. Di Caprio*
Une merveilleuse histoire du temps	*J. Marsh*	*E. Redmayne*

SÉANCES

Cinéma	Horaire	Titre
Le grand club	*18h*	*Titanic*
Le Palace	*18h30*	*Terminator*
Le Palace	*20h*	*Inception*
La Plage	*19h*	*Inception*

A Domaine et signature

Domaine. On se donne un **domaine** D, c'est-à-dire un ensemble qui contient tous les éléments pouvant apparaître dans un table. Pour l'exemple précédent, on aurait pu prendre l'ensemble des chaîne de caractère.

Signature. À chaque table on peut faire correspondre un ensemble d'**attributs**. Par exemple, les attributs de la table `FILMS` sont :

Titre, **Réalisateur**, **Acteur**

On appelle alors **schéma de relation** la spécification d'une table et de ses attributs, et une **signature** est un ensemble de schémas de relation.

Exemple 22.2 *Pour la table* `FILMS`*, le schéma de relation est :*

`FILMS`*[***Titre**, **Réalisateur**, **Acteur***]*

B Tuple

De manière informelle, un tuple correspond à une ligne de la table.

Définition 22.2 *Soit $R[U]$ un schéma de relation. Un* **tuple** *sur $R[U]$ est une fonction $t : U \to D$. On notera le tuple par :*

$$u = \langle U_1 : t(U_1), ..., U_2 : t(U_2) \rangle$$

où $U = \{U_1, ..., U_n\}$. On autorisera à noter $u = \langle t(U_1), ..., t(U_n) \rangle$ s'il n'y a aucune ambiguïté.

Exemple 22.3 *Un tuple sur* `FILMS`*[***Titre**, **Réalisateur**, **Acteur***] est :*

$\langle$ **Titre** *: The Dark Knight,* **Réalisateur** *: C. Nolan,* **Acteur** *: C. Bale*$\rangle$

On remarque que ce tuple n'apparaît pas dans la table `FILMS`*, mais cela reste un tuple pour le schéma correspondant.*

C Base de données

Définition 22.3 (Table) *Un ensemble fini de tuples sur un schéma de relation $R[U]$ s'appelle une* **table** *(ou* **relation***) de $R[U]$. S'il n'y a pas d'ambiguïté, on peut la noter R.*

La définition classique d'une base de donnée est alors simplement un **ensemble de tables**. C'est là que la vision logique intervient, puisqu'on peut voir une base de données comme un modèle.

Définition 22.4 (Base de donnée) *Soit S une signature. une* **base de donnée** *est un modèle $\mathcal{M} = (D, (d^{\mathcal{M}})_{d \in D}, (R^{\mathcal{M}})_{R[U] \in S})$ tel que :*

- *pour tout $d \in D$, $d^{\mathcal{M}} = d$;*
- *pour tout schéma de relation $R[U] \in S$, $R^{\mathcal{M}}$ est une table de $R[U]$.*

En pratique. On peut interagir avec une base de données, pour lire des données qui vérifient certaines conditions. Dans les SGBD, cela se fait à l'aide de requêtes, dans un langage de programmation spécial, le plus répandu de ces langages est SQL.

Notons qu'aujourd'hui on observe un changement de paradigme vers des bases de données qui utilisent des graphes plutôt que des relations. Par exemple, le système Neo4j qui utilise le langage Cypher, permettant de faire des requêtes sur des graphes.

II Concevoir une base de données relationnelle

Exemple 22.4 *Un site de cinéphile souhaite gérer une base de donnée contenant l'ensemble des films actuellement au cinéma dans leur ville.*

Il y a une infinité de manières de représenter ce problème sous forme relationnelle. Certaines peuvent présenter des problèmes de redondance, ou d'intégrité. Comment trouver une bonne manière de représenter ces données ?

A Clés primaires, clés étrangères

Clé primaire. On doit avoir un moyen de spécifier comment les tuples sont distingués au sein d'une même table. Les valeurs d'un tuple doivent permettre d'identifier uniquement chaque tuple.

Les clés primaires et étrangères sont deux concepts permettant d'imposer certaines contraintes, dites "d'intégrité", dans un schéma relationnel.

Définition 22.5 *Soit $R = \{t_1, ..., t_n\}$ une table sur le schéma de relation $R[U]$.*

- *Une* **superclé** *est un sous ensemble d'attributs $I \subseteq U$ tel que si $t_1, t_2 \in T$, alors $t_1.I \neq t_2.I$.*
- *Une* **clé primaire** *est une superclé minimale pour l'inclusion.*

Les clés primaires sont un moyen utile d'éviter les doublons : lors de l'insertion d'un nouvel enregistrement, un système de gestion peut vérifier que cela ne crée pas de doublon au niveau de la clé primaire de la table.

Clé étrangère. On peut ajouter une nouvelle contrainte à notre base de donnée. Si une de ses tables contient un certain attribut, on peut forcer ces valeurs à apparaître dans l'autre table.

Définition 22.6 *Soient R_1, R_2 deux tables sur les schémas $R_1[U_1]$ et $R_2[U_2]$, tels que R_1 possède parmi ses attributs une clé primaire pour R_2. Cet attribut ou ensemble d'attributs est appelé* **clé étrangère** *de T_1 référençant T_2.*

Les clés étrangères permettent de faire le lien entre différentes relations, et garantit l'intégrité référentielle du schéma. Cela signifie que lorsque l'on insère un enregistrement x dans R_1, on doit vérifier qu'il existe dans R_2 un enregistrement dont la valeur pour la clé primaire correspond à la valeur de x pour la clé étrangère .

Exemple 22.5 *Pour la table* `FILMS`*, l'attribut* **Titre** *est une clé primaire, et c'est une clé étrangère de* `FILMS` *référençant la table* `SÉANCES`*.*

B Modèle Entité-Association

Le modèle entité-association est un modèle conceptuel permettant de visualiser des données structurées. Lors de la conception d'une base de donnée, on commence généralement par représenter les données dans ce modèle. Puis, on peut passer du modèle EA au modèle relationnel, en utilisant des clés primaires et étrangères qui correspondront aux contraintes réelles du problème perçues par le concepteur de la base.

Exemple 22.6 *Schéma UML de l'exemple filé*

Définition 22.7 — *Une* **entité** *est un objet possédant un nom et des attributs.*

— *Une* **association** *est un lien entre deux entités, reflétant un lien conceptuel. On représente une association par une ligne tracée entre deux entités, avec le nom de l'association indiqué.*

— *Une association possède deux* **arités**, *chacune contenant deux nombres, qui correspondent respectivement au maximum et au minimum d'entités avec lesquelles une entité peut être en lien via l'association.*

— *Si une entité peut être entièrement déterminé par un sous-ensemble de ses attributs, on souligne ces attributs pour le mettre en évidence.*

Exemple 22.7 *Exemple filé*

C Structure de donnée.

En pratique, une base de donnée est stockée sur un disque externe, et les opérations les plus coûteuses en temps sont alors la lecture et l'écriture sur le disque. Ainsi que le SGBD optimise l'accès aux tuples d'une base de données, il est judicieux de recourir à une structure de donnée arborescente avec une ramification importante. C'est typiquement ce qu'offre la structure de **B-arbres**.
Développement 3. Présentation des B-arbres et complexité de l'algorithme de recherche.

D Langages de requêtes relationnelles

Un **langage de requêtes** est un langage permettant à un utilisateur d'accéder à des informations dans une base de donnée. Il y a trois types de langages de requêtes : impératif, fonctionnel ou déclaratif.

Langage impératif. Dans un tel langage, l'utilisateur décrit une séquence d'opérations pour calculer le résultat voulu.

Langage fonctionnel. Dans un tel langage ; le calcul est exprimé par l'évaluation d'une fonction pouvant opéré sur les les données ou sur le résultat d'un autre appel de fonction.

Langage déclaratif. Dans un tel langage, l'utilisateur décrit les informations désirées sans donner une séquence spécifique d'opération ou une fonction. L'information est le plus souvent décrite en utilisant une forme de logique mathématique. C'est le système de gestion de base de donnée (SGBD) qui gère comment obtenir cette information.

On présente ici deux langages de requête « pure » :

— l'**algèbre relationnelle** qui est un langage fonctionnel et forme la base théorique du langage `SQL` ;

— le **calcul relationnel**, qui lui est déclaratif.

III Algèbre relationnelle

L'algèbre relationnelle consiste en un ensemble d'opérations qui prend en entrée une ou plusieurs tables en entrée, et produit une nouvelle table en sortie.

A Opérations et requêtes

On présente ici d'abord les opérations de l'algèbre relationnelle informellement :

— la **sélection** : permet d'extraire certaine ligne via une formule booléenne ;

— la **projection** : permet d'extraire certaines colonnes ;

— les opérations ensemblistes : **produit cartésien**, **union** et **différence**.

Définition 22.8 (Requête de l'algèbre relationnelle) *On définit les requêtes de l'algèbre relationnelle par induction :*

- *le tuple* $\langle j : d\ rangle$;
- *toute table* R ;
- $\sigma_b(q)$ *avec* b *de forme* $j_1 = j_2$ *ou* $j_1 = a$;
- $\pi_{j_1,...,j_k}(q)$;
- $q \times q'$;
- $q \cup q'$;
- $q \backslash q'$

sont des requêtes, où q *et* q' *sont des requêtes,* $j_1, ..., j_k$ *des attributs et* a *une constante. Pour l'union et la différence,* q *et* q' *doivent avoir les mêmes attributs.*

Exemple 22.8 *La requête « Quels films ont-été réalisés par C. Nolan ? », on écrit :*

$$\pi_{\textbf{Titre}}\left(\sigma_{\textbf{Réalisateur} = \textit{C. Nolan}}(\texttt{FILMS})\right)$$

B Sémantique

Définition 22.9 *Étant donné une base de donnée* $\mathcal{M}$ *et une requête* q *de l'algèbre relationnelle, on définit la sémantique de* q *(ou la réponse) par induction sur* q *:*

- $[\![\langle j : d\rangle]\!]^{\mathcal{M}} = \{\langle j : d\rangle\}$;
- $[\![R]\!]^{\mathcal{M}} = R^{\mathcal{M}}$;
- $[\![sigma_b(q)]\!]^{\mathcal{M}} = \{u \in [\![q]\!]^{\mathcal{M}} | b(u)$ *est vrai*$\}$;
- $[\![\pi_{j_1,...,j_k}(q)]\!]^{\mathcal{M}} = \{\langle u_{j_1}, ..., u_{j_k}\rangle | u \in [\![q]\!]^{\mathcal{M}}\}$
- $[\![q \times q']\!]^{\mathcal{M}} = \{\langle u, u'\rangle | u \in [\![q]\!]^{\mathcal{M}}, u \in [\![q']\!]^{\mathcal{M}}\}$
- $[\![q * q']\!]^{\mathcal{M}} = [\![q]\!]^{\mathcal{M}} * [\![q']\!]^{\mathcal{M}}$ *pour* $* \in \{\cup, \backslash\}$.

Exemple 22.9 *La réponse à la requête précédente est alors la table :*

Titre
Inception

Requête satisfaisable. Une requête q est alors dit satisfaisable s'il existe une base de donnée $\mathcal{M}$ pour laquelle $[\![q]\!]^{\mathcal{M}}$ soit non vide.

Algèbre SPC. On définit l'algèbre SPC comme l'algèbre relationnelle restreinte aux requêtes ne contenant ni union, ni différence.

IV Calcul relationnel

La logique du premier ordre appliquée aux bases de données s'appelle le **calcul relationnel**. On écrit une requête comme une formule, où les variables libres sont les éléments à trouver. Comme on nomme les attributs, une formule atomique est de la forme $R(u)$ où $R[U]$ est un schéma de relation et $u : U \to D \cup X$ où D est un domaine et X l'ensemble des variables.

Exemple 22.10 `FILMS(⟨ Titre :` x`, Réalisateur : C. Nolan, Acteur :` y `⟩)` *est une formule atomique. On notera* `FILMS`$(x,$ *C. Nolan,* $y)$.

A Requête du calcul relationnel

Requête. On peut alors simplement définir une **requête du calcul relationnel** comme une formule,c conforme à la syntaxe suivante :

$$R(x_1, ..., x_n)|\neg\varphi|(\varphi \wedge \psi)|\exists x\varphi$$

Exemple 22.11 *La requête « Quels fils ont-été réalisés par C. Nolan ? », on écrit :*

$$\exists y, \texttt{FILMS}(x, \text{C. Nolan}, y)$$

La seule variable libre est x, et correspond à l'attribut **Titre** *recherché.*

Sémantique. On définit maintenant la sémantique d'une telle formule, correspondant à la réponse à la requête.

Définition 22.10 (Sémantique d'une requête du calcul relationnel) *La* **sémantique** *d'une requête φ sur une base de donnée $\mathcal{M}$, noté $[\![\varphi]\!]^{\mathcal{M}}$, est définit inductivement sur les formules :*

- $[\![R(x_1, ..., x_n)]\!]^{\mathcal{M}} = R^{\mathcal{M}}$;
- $[\![(x = y)]\!]^{\mathcal{M}} = \{\langle d, d\rangle | d \in D\}$;
- $[\![(x = a)]\!]^{\mathcal{M}} = \{\langle a\rangle\}$;
- $[\![\psi(x_1, ..., x_n) \wedge \theta(y_1, ..., y_n)]\!]^{\mathcal{M}} = [\![\psi(x_1, ..., x_n)]\!]^{\mathcal{M}} \bowtie_{(i,j)|x_i=y_j} [\![\theta(y_1, ..., y_n)]\!]^{\mathcal{M}}$;
- $[\![\neg\psi(x_1, ..., x_n)]\!]^{\mathcal{M}} = (D^{\mathcal{M}})^n \backslash [\![\psi(x_1, ..., x_n)]\!]^{\mathcal{M}}$;
- $[\![\exists\psi(x_1, ..., x_n)]\!]^{\mathcal{M}} = \pi_{\neq i}[\![\psi(x_1, ..., x_n)]\!]^{\mathcal{M}}$;

où $\bowtie$ et la jointure définit ci-dessous, et $\pi_{\neq i}$ est la projection sur toutes les composantes différentes de i.

Étant donné deux tables R_1 d'arité n et R_2 d'arité m , on définit la $(k - l)$ jointure de R_1 et R_2 :

$$R_1 \bowtie_{k,l} R_2 = \pi_{\neq n+l}(\{(d_1, ..., d_n, d'_1, ..., d'_m) \in D^{n+m} | (d_1, ..., d_n) \in R_1, (d'_1, ..., d'_m) \in R_2, d_k = d'_l\})$$

On étend alors cette définition pour un ensemble de couples $(k_1, l_1), ..., (k_N, l_N)$.

Cette sémantique est cohérente avec la sémantique classique en logique du premier ordre.

Théorème 22.1 (Cohérence de la sémantique relationnelle)
pour toute structure $\mathcal{M}$, et toute formule $\varphi(x_1, ..., w_k)$, on a

$$(d_1, ..., d_k) \in [\![\varphi(x_1, ..., x_k)]\!]^{\mathcal{M}}$$

si, et seulement si,

$$\mathcal{M}, [x_1 := d_1], ..., [x_k := d_k] \models \varphi(x_1, ..., x_k).$$

Exercice 22.1 *Déduire du théorème précédent que pour toute structure $\mathcal{M}$ et toute formule close φ, on a $\mathcal{M} \models \varphi$ si, et seulement si, $[\![\varphi]\!]^{\mathcal{M}} \neq \emptyset$.*

V Indépendance du domaine

Domaine. Le domaine peut clairement être infini (comme l'ensemble des chaînes de caractères). Deux problèmes peuvent alors apparaître :

1. la réponse à une requête peut être infini ;
2. la réponse peut dépendre du domaine.

Ces problèmes apparaissent notamment avec la négation, le disjonction et le quantificateur universel. On va donc se tourner vers un modèle plus faible du calcul relationnel, mais qui permet d'éviter ces problèmes.

Indépendance du domaine. Une requête φ est indépendante du modèle si, pour toutes bases $\mathcal{M}$ et $\mathcal{M}'$ qui ne diffèrent que de leur domaine, on a $[\![\varphi]\!]^{\mathcal{M}} = [\![\varphi]\!]^{\mathcal{M}'}$.

Exemple 22.12 *On prend $D^{\mathcal{M}} = \{1\}$ et $D^{\mathcal{M}'} = \{1, 2\}$, et on considère la table $R^{\mathcal{M}} = R^{mathcal M'} = \{\langle 1, 1\rangle\}$. Il suffit alors de prendre la requête $\forall y, R(x, y)$ qui s'interprète comme un élément de la première colonne en relation avec tous les éléments du domaine.*

Domaine actif. Pour éviter les réponses infini, on se restreint au **domaine actif**. Le domaine actif de $\mathcal{M}$ (resp. φ) est l'ensemble des éléments de D qui sont aussi dans $\mathcal{M}$ (resp. φ).

A Calcul relationnel conjonctif

On va donc créer un sous-ensemble du calcul relationnel où les requêtes sont indépendantes du domaine.

Définition 22.11 *Soit S une signature. Une formule du calcul conjonctif est une requête ne contenant que les connecteurs $\wedge$ et $\exists$, et où on interdit les formules $(x = y)$ avec x et y libres n'apparaissant dans aucune autre relation R.*

Exemple 22.13 *Les formules $R(x, y) \wedge (x = a)$ ou encore $\exists y R(x, y, z) \wedge (x = y)$ sont des formules du calcul conjonctif, mais pas $R(x, y) \wedge (z = x)$ car z n'apparaît que dans l'égalité.*

Théorème 22.2 *Les requêtes du calcul conjonctif sont finies (leurs réponses sur toute abse de donnée est un ensemble fini), indépendantes du domaine et satisfaisables.*

VI Théorème de Codd

Edgar Frank Codd est l'inventeur du concept de base de données relationnelle, et à ce titre, lauréat du prix Turing en 1981. On lui doit notamment l'équivalence entre les deux langages de requêtes vu.

Théorème 22.3 (Théorème de Codd conjonctif) *Il y a équivalence entre les requête du calcul conjonctif et les requêtes satisfaisables de l'algèbre SPC.*

Développement 19. Preuve partielle du théorème ci-dessus.

Théorème 22.4 (Théorème de Codd) *Restreint au domaine actif, le calcul relationnel et l'algèbre relationnelle sont équivalents.*

Leçon 23

Requêtes en langage SQL.

Auteur·e·s: Marin Malory
Niveau : L1
Pré-requis : introduction au modèle relationnel et bases de données relationnels (algèbre relationnelle non nécessaire)
Références : [Silberschatz et al., 2020]

I Le langage SQL

Les serveurs de données relationnels présentent aujourd'hui une interface externe sous forme d'un langage de recherche et de mis à jour, permettant de spécifier les ensembles de données en fonction de certaines valeurs. Les opérations directement utilisables par les usagers sont en général celles des langages dits assertionnels, comme le SQL ou **Structured Query Langage**. Aujourd'hui, le langage SQL est normalisé (par l'organisme ANSI) et constitue le standard d'accès aux bases de données relationnelles. De manière générale, SQL comme les autres langages qui ont été proposés comportent quatre opérations de bases : le **recherche**, l'**insertion**, la **suppression** et la **modification**.

II Recherche de données

On utilisera l'exemple des tables suivantes.

Exemple 23.1 *Voici un exemple de trois tables, ou relations, appelés* `Films` *et* `Séances`.

`Films`

titre	réalisateur	acteur
Titanic	*J. Cameron*	*L. Di Caprio*
Terminator	*J. Cameron*	*A. Schwarzenegger*
Inception	*C. Nolan*	*L. Di Caprio*
Une merveilleuse histoire du temps	*J. Marsh*	*E. Redmayne*

`Séances`

cinéma	horaire	titre	prix
Le grand club	*18h*	*Titanic*	*8*
Le Palace	*18h30*	*Terminator*	*9*
Le Palace	*20h*	*Inception*	*9*
La Plage	*19h*	*Inception*	*7*

A La projection

La *projection* d'une table en vue de l'obtention d'un ensemble de colonnes de cette table se fait via l'ordre `SELECT`.

Exemple 23.2 *Si on veut récupéré la table des titres et des noms des réalisateurs, on utilisera la requête suivante*

```
SELECT titre, réalisateur FROM Films
```

Remarque 23.1 *Lorsqu'on veut sélectionner toutes les colonnes, on peut remplacer l'ensemble des noms par le caractère ∗.*

Avec une simple requête `SELECT ... FROM ...`, on n'élimine pas les doublons. Le mot clé `DISTINCT` permet de les supprimer.

Exemple 23.3 *On veut la table des noms des réalisateurs sans doublons, on utilisera alors la requête :*

```
SELECT DISTINCT réalisateur FROM Films
```

B Restrictions-Sélection

On peut faire des *restriction* (ou *sélection*, mais le nom est trompeur) à l'aide du mot-clé `WHERE`. Là où les projections permettent de sélectionner des colonnes , les restrictions permettent de sélectionner des lignes de la table.

Exemple 23.4 *On cherche les séances ayant un prix inférieurs à 8€ :*

```
SELECT * FROM Séances
WHERE prix <= 8 ;
```

On peut alors faire des restrictions plus complexes à l'aide des comparateurs arithmétiques $<$,$>$, $<=$, $>=$, $=$, $!=$ et des opérateurs logiques `AND`, `OR` et `NOT`.

Il existe encore d'autres prédicats permettant des requêtes encore plus complexe :

- `IN` peut être utilisé dans des conditions de la forme `expression IN (valeur_1,...,valeur_n)` ou encore `expression IN (sous_requête)` ;
- `BETWEEN` peut être utilisé dans des conditions de la forme `expression BETWEEN valeur_1 AND valeur_2` ;
- `LIKE` permet de faire un filtrage, voir exemple ci-dessous

Exemple 23.5 *On veut récupérer la table des films ayant un titre commençant par 'T', on utilise la requête suivante :*

```
SELECT * FROM Films
WHERE titre LIKE 'T%' ;
```

Remarque 23.2 *Le caractère '%' signifie ici « n'importe quelle chaîne de caractère » , on pourrait aussi utiliser le caractère '_' qui signifie « n'importe quel caractère », mais on contraint alors la taille de la chaîne de caractère.*

C Tri et présentation des résultats

`SQL` permet de trier les résultats via le mot-clé `ORDER BY`. Les mots-clés `ASC` et `DESC` permettent de préciser si on trie dans l'ordre croissant ou décroissant.

Exemple 23.6 *On veut trier les séances par prix croissant, et en cas d'égalité par ordre alphabétique du titre du film :*

```
SELECT * FROM Séances
ORDER BY prix ASC, titre ASC ;
```

Remarque 23.3 *L'ordre des éléments après un* `ORDER BY` *compte. Dans l'exemple ci-dessus, on trie d'abord les éléments par prix croissant et c'est seulement en cas d'égalité qu'on trie par nom croissant.*

Il est parfois utile de ne pas récupérer tous les résultats, mais seulement les n premiers. On utilisera pour cela le mot-clé `LIMIT`. Si on veut décaler les le résultat (par exemple si on ne veut pas récupérer les m premiers), on utilisera le mot-clé `OFFSET`

D Jointures

Les **jointures** en SQL permettent d'associer plusieurs tables via une même requête, en associant certains attributs de chaque table.

Une jointure est donc un sous-ensemble du produit cartésien entre deux tables, obtenue via la requête `SELECT * FROM table1,table2,... ;`.

Remarque 23.4 *Dès qu'on combine plusieurs tables, on peut utiliser le nom des tables comme préfixe pour sélectionner un attribut spécifique. Par exemple, on pourra avoir une commande de la forme* `SELECT table1.id, table2.id FROM table1,table2 ;`.

On peut alors utiliser le mot-cl" `JOIN` pour resteindre ce produit cartésien selon une condition. La syntaxe est alors `SELECT ... FROM table1 JOIN table2 ON condition ;`.

Exemple 23.7 *On veut l'ensemble des séances avec le réalisateur et l'acteur principal en plus pour chaque films.*

```
SELECT *
FROM Séances
JOIN Films
ON Séances.titre = Films.titre ;
```

Remarque : *Cette commande peut aussi s'écrire de la manière suivante :*

```
SELECT *
FROM Séances, Films
WHERE Séances.titre = Films.titre ;
```

Il est aussi possible de réaliser ce que l'on appelle des *jointures externes*, permettant de conserver tous les éléments d'une table même s'il aurait été éliminer par une jointure classique.

Exemple 23.8 *Si on remplace le* `JOIN` *par un* `LEFT JOIN` *dans la commande précédente, on aura toutes les séances, même si on ne dispose pas des informations concernant ce film dans la table* `Films`*.*

Il existe ainsi plusieurs types de jointures :

— `LEFT JOIN` : jointure externe gauche ;

— `RIGHT JOIN` : jointure externe droite ;

— `FULL JOIN` : renvoie la ligne si la condition est vraie sur au moins une des deux tables.

— `SELF JOIN` : jointure d'une table avec elle-même ;

— `NATURAL JOIN` : jointure naturelle si deux attributs sont les mêmes, fait la jointure automatiquement dessus.

III Requêtes avancées

A Fonctions statistiques

Au delà de la simple recherche de données, des possibilités de calcul de fonctions existent. Les fonctions implantées sont :

— `COUNT`, qui permet de compter le nombre de valeur d'un ensemble ;

— `SUM` qui permet de sommer les valeurs d'un ensemble ;

— `AVG` qui permet de calculer la valeur moyenne d'un ensemble ;

— `MAX` (resp. `MIN`) qui permet de calculer la valeur maximum (resp. minimum) d'un ensemble.

Exemple 23.9 *Par exemple, si on veut compter le prix moyen des séances pour le film Titanic, on utilisera*

```
SELECT AVG(prix)
FROM Séances
WHERE titre = "Titanic" ;
```

Ces fonctions peuvent être utilisées dans la clause `SELECT`, et sont aussi utilisables pour effectuer des calculs d'agrégats.

B Agrégats

Un **agrégat** est un partitionnement horizontal d'une table en sous-table en fonction des valeurs d'un ou plusieurs attributs de partitionnement, suivi de l'application d'une fonction de calculs à chaque attribut des sous-tables tables obtenues.

En SQL, le partitionnement s'exprime via la clause `GROUPE BY <spécifications colonnes>`.

Exemple 23.10 *On souhaite afficher le nombre de séances par cinéma :*

```
SELECT cinéma, COUNT(titre)
FROM Séances
GROUP BY cinéma ;
```

La clause `HAVING` permet quant à elle d'exprimer une restriction sur les lignes de la table obtenue avec les fonctions de calculs. Ainsi,

— les tests sur les colonnes simples s'opèrent dans la clause `WHERE` ;

— les tests sur les fonctions de regroupement se font dans la clause `HAVING`.

Exemple 23.11 *On souhaite afficher les cinémas proposant au moins 2 séances :*

```
SELECT cinéma
FROM Séances
GROUP BY cinéma
HAVING COUNT(titre) > 1 ;
```

C Exercice

On résout ici un problème à l'aide de requêtes avancées, sur une base de données sur des entreprise.

L'idée est notamment la syntaxe `WITH ... AS ...`, permettant ainsi de simplifier certaines requêtes.
Développement 19. Résolution d'un exercice avancé de SQL.

IV La définitions des schémas

SQL permet de définir des schémas de bases de données composés de **tables** et de **vues**.

A Création de tables

SQL permet la créer des relations sous forme de table via la requête `CREATE TABLE nom_table (élément1, élément2,...)`, où chaque éléments est de la forme `nom_element Complement` où le complément peut donner des informations, sur le type notamment.

Exemple 23.12 *On peut créer la table `Séances` via la requête suivante :*

```
CREATE TABLE Séances (
        cinéma CHAR(20),
        horaire CHAR(20),
        titre CHAR(20)
        prix INT )
```

Remarque 23.5 *À la création d'une table, on peut ajouter des* **contraintes d'intégrité**, *par exemple en spécifiant l'unicité de l'attribut (syntaxe `UNIQUE` ou `PRIMARY KEY`) ou le fait que la donnée ne peut être manquante (syntaxe `NOT NULL`).*

Remarque 23.6 *À la place d'une table, on peut créer une* **vue** *via la commande `CREATE VIEW`. Celle-ci sont juste des tables d'affichage et ne peuvent pas être mis à jour.*

Indexation. Il est possible d'organiser nos données via une indexation sur une ou plusieurs colonnes, permettant ainsi d'accélérer certaines requêtes. Cela fonctionne via la syntaxe `CREATE INDEX index_nom ON table (colonne1, colonne2);`. Une structure de donnée classique en indexation est un B-arbre.
Développement 3. Présentation des B-arbres.

Leçon 24

Exemples d'algorithmes d'apprentissage supervisés et non supervisés.

Auteur·e·s: Marin Malory
Niveau : L2-L3
Pré-requis : Algorithmique, Probabilités
Références : [Biernat et al., 2015], [Shalev-Shwartz and Ben-David, 2014]

Introduction

Définition 24.1 (Apprentissage machine, Tom Mitchell, 1997) *On dit qu'un programme apprend via l'entraînement E pour un ensemble de tâches T et une mesure de performance P si sa performance sur T mesuré par P s'améliore après E.*

On considère deux familles de l'apprentissage machine :

1. l'**apprentissage supervisé** : on dispose d'espaces $\mathcal{X}$ d'entrée et $\mathcal{Y}$ de sortie, et d'un ensemble d'exemples $(x_i, y_i) \in \mathcal{X} \times y$, le but étant de construire une fonction $\hat{h}_n : \mathcal{X} \to y$.
2. l'**apprentissage non supervisé** : on dipose seulement de l'espace $\mathcal{X}$, et on fait du clustering (classification en français mais il y a une ambiguïté avec l'apprentissage supervisé). Le but est alors de construire une partition de $\mathcal{X}$.

Il existe d'autres familles, comme l'**apprentissage par renforcement** qui consiste à maximiser un critère d'utilité par expérience successives.

I Apprentissage supervisé

On définit formellement un problème d'apprentissage supervisé.

Données :

— **Domaine** : ensemble quelconque $\mathcal{X}$, souvent muni d'une distance d ;
— **Ensemble d'arrivée** : ensemble quelconque $\mathcal{Y}$;
— **Ensemble d'entraînement** : $S = \{(x_1, y_1), ...(x_n, y_n)\}$ une suite finie de $\mathcal{X} \times \mathcal{Y}$.

Sortie : Une fonction $h : \mathcal{X} \to \mathcal{H}$.

Exemple 24.1 (Problème de météo) *On considère un problème où, en fonction de la température et de l'humidité, on essaye de savoir s'il y a du soleil : On a alors :*

— $\mathcal{X} = \mathbb{R}^2$ *(température* $\times$ *humidité) ;*
— $\mathcal{Y} = \{soleil, gris\}$ *;*
— S *est un ensemble de mesures météorologiques de la forme* $((T, H), \Delta)$ *où* T *et* H *sont respectivement une température et une mesure d'humidité, et* Δ *est choisi en fonction s'il y a des nuages.*

Taxiconomie. On distingue plusieurs problèmes :

— si $\mathcal{Y}$ est discret (comme dans l'exemple ci-dessus), on parle de **classification** ;

— si $\mathcal{X}$ est continue, on parle de **regression**.

On supposera dans tout le reste de cette partie que $\mathcal{X} = \mathbb{R}^d$ pour $d > 0$, et notre ensemble d'entraînement par une matrices $X \in \mathbb{R}^{n \times d}$.

A Algorithmes de régression

Dans cette partie, on suppose $d = 1$. Notre ensemble d'apprentissage est donc un ensemble de points sur un plan.

Régression linéaire univariée. L'idée derrière cette méthode est simple, on cherche à trouver la droite la plus proche de nos points.

On pose ici une **fonction hypothèse** $h : x \mapsto \theta_0 + \theta_1 x$ où θ_0 et θ_1 sont des paramètres à optimiser, de telle manière à ce que $h(x_i)$ soit proche de y_i pour tout $1 \leq i \leq n$. Pour cela, on cherche à minimiser la fonction :

$$J(\theta_0, \theta_1) = \frac{1}{2n} \sum_{i=1}^{n} (h(x_i) - y_i)^2$$

On peut alors utiliser des algorithmes d'**optimisation** pour minimiser cette fonction (qui est convexe par ailleurs), comme la descente de gradient.

> **Remarque 24.1** *On peut généraliser cette méthode pour d quelconque, on parle alors de régression linéaire multivariée.*

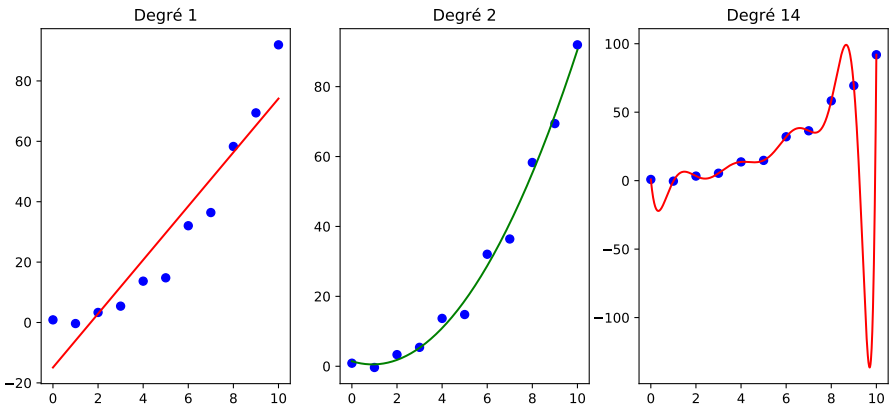
Dans certains cas, notre modèle n'est pas assez précis, on parle de **sous-apprentissage** (voir figure ci-dessous).

Régression polynomiale. On va ici présenter une méthode similaire à la précédente, sauf que notre fonction hypothèse sera une fonction polynomiale. La fonction hypothèse est donc de la forme :

$$h : x \mapsto \sum_{i=0}^{k} a_i x^i$$

où k est un paramètre à fixer, et les coefficients $a_0, ..., a_k$ sont à optimiser.

Attention, si k est trop grand, on perd la robustesse de notre modèle qui ne généralise plus correctement : on parle de **sur-apprentissage** (voir figure ci-dessous).



Remarque 24.2 *On remarque qu'il faut trouver un compromis entre l'***erreur d'apprentissage** *(erreur du modèle sur l'ensemble d'entraînement) et l'***erreur de prévision**. *On parle de* **compromis biais-variance**.

B Algorithme des k plus proches voisins

L'algorithme des k plus proches voisins (k-NN) est l'un des plus simples en apprentissage supervisé. L'idée est de garder en mémoire l'entièreté du jeu d'entraînement, et de prédire la valeur en fonction des données les plus proches.

Pour $x \in \mathcal{X}$ et $S = (x_1, y_1), ..., (x_n, y_n)$, on note $\pi_1(x), ..., \pi_n(x)$ la permutation de $(1, ..., n)$ telle que pour tout $1 \leq i < n$,

$$d(x, x_{\pi_i(x)}) \leq d(x, x_{\pi_{i+1}(x)})$$

Autrement dit, on range les données par distance croissante à x.

Algorithme 24.1 : k-NN(S,x)

Données : $S = \{(x_1, y_1), ..., (x_n, y_n)\} \subset \mathcal{Y} \times \mathcal{Y}$, $x \in \mathbb{R}^d$
. $y \leftarrow \text{argmax}_{y \in \mathcal{Y}} \sum_{i=1}^{k} \mathbf{1}\{y_{\pi_i(x)} = y\}$;

Pour $k = 1$, l'algorithme retourne simplement $y_{\pi_1(x)}$.

Remarque 24.3 *Lorsque $\mathcal{Y}$ est continue, on peut, à la place de retourner la majorité, retourner la moyenne, voir la moyenne pondéré par l'inverse de la distance.*

C Arbres de décision

Un arbre de décision est un prédicteur $h : \mathcal{X} \to \mathcal{Y}$ qui prédit la valeur de x en traversant un arbre de la racine jusqu'à une feuille.

Exemple 24.2 (Le fruit est-il mûre ?) *Un arbre de décision pourrait suivre le raisonnement suivant : si la couleur est vive, le fruit est mûre. Sinon, le fruit est mûre si, et seulement s'il est mou.*

Pour construire un arbre de décision avec notre jeu d'entraînement, l'idée la plus simple est que chaque nœud soit un seuil pour une des caractéristiques.

On présente ici un algorithme récursif classique, appelé ID3, permettant de construire un classifieur dans le cas où $\mathcal{X} = \{0, 1\}^d$ et $\mathcal{Y} = \{0, 1\}$.

Algorithme 24.2 : ID3(S, A)

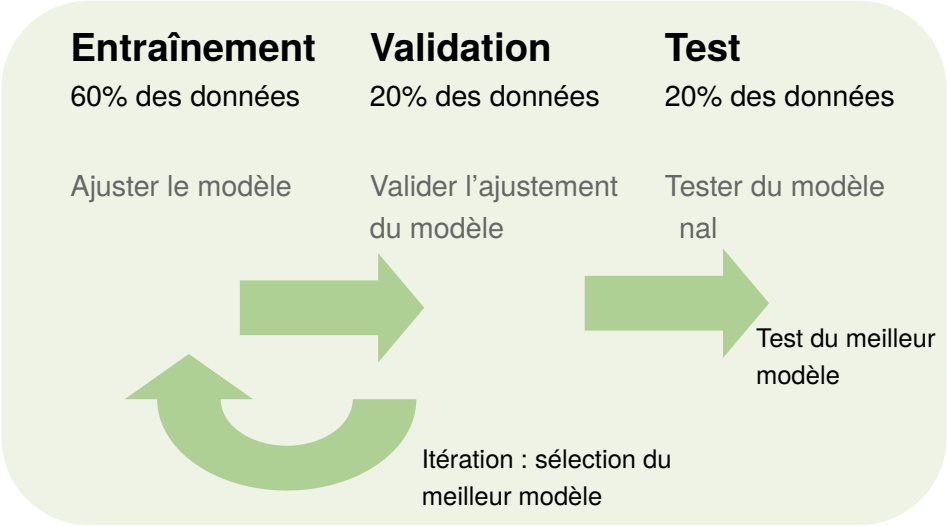
Données : ensemble d'entraînement S, sous-ensemble de caractéristiques $A \subset [d]$
si $\forall (x, y) \in S, y = 1$ **alors**
  retourner *feuille* 1 ;
si $\forall (x, y) \in S, y = 0$ **alors**
  retourner *feuille* 0 ;
si $A = \emptyset$ **alors**
  retourner *feuille (majorité* 0 *ou* 1 *dans* S*)* ;
$j \leftarrow \text{argmax}_{i \in A} \texttt{Gain}(S, i)$;
$T_1 \leftarrow$ ID3$(\{(x, y) \in S | x_j = 1\}, A \backslash \{j\})$;
$T_1 \leftarrow$ ID3$(\{(x, y) \in S | x_j = 0\}, A \backslash \{j\})$;
retourner $((x_j = 1?), T_1, T_2)$

Comment choisir la fonction de gain ? On veut une fonction qui permet de mesurer à quelle point notre coupe enlève du désordre, on peut prendre la fonction de Gini ou l'entropie.

> **Remarque 24.4** *Ici on a un long temps d'apprentissage mais la prédiction est rapide (à l'inverse des k plus proches voisins). Il est aussi facilement interprétable.*

D Au-delà des algorithmes

La nécessité de diviser nos données. Puisque dans notre cas, on n'a un choix limité de donnée pour tester notre algorithme une division s'impose. En effet, il nous faut des données pour **entraîner**, pour **optimiser** les paramètres (**validation**) et enfin pour **tester** notre algorithme final. La méthode la plus simple consiste à couper notre jeu de donnée en 3 (60%, 20%, 20%) (voir figure ci-dessous)



Il existe d'autres méthodes, comme la validation croisée.

Maintenant, on a un prédicteur ou un classifieur, **comment estimer sa performance ?** On s'intéresse ici au cas des classifieur ($\mathcal{Y}$ discret).

Approche empirique

Définition 24.2 (Matrice de confusion) *Matrice M de taille $N \times N$ où N est le nombre de label.Les colonnes sont les prédictions et les lignes la réalité. Une bonne matrice de confusion donne des gros nombres sur la diagonal et peu ailleurs.*

On déduit de cette matrice la précision, qui est le nombre de bonnes prédictions sur le nombre totale.

Approche théorique : apprentissage statistique. On peut mettre un modèle probabiliste sur nos données : les données (x_i, y_i) sont des réalisations de variables aléatoires $(X_i, Y_i)_{1 \leq i \leq n}$ et les (X_i, Y_i) sont iid selon la loi (X, Y). On s'intéresse surtout à la probabilité $P_{Y|X}$.

On ne connaît pas la loi $P_{(X,Y)}$, car sinon on aurait un prédicteur simple, le **prédicteur de Bayes** :

$$h^* : x \mapsto \text{argmax}_y P(Y = y | X = x)$$

qui, étant donné un point dans l'espace, retourne le y qui maximise la probabilité d'avoir y sachant qu'on a la donnée x. On peut effectivement montrer que c'est le meilleur prédicteur.

Définition 24.3 (Performance d'un classifieur) *Étant donné un classifieur $h : \mathcal{X} \to \mathcal{Y}$ et une distribution (X,Y) sur $\mathcal{X} \times \mathcal{Y}$, on pose $l_P(h) = P_{(X,Y)}(h(X) \neq Y)$ la performance de h sur la distribution (X,Y).*

On a alors :

Proposition 24.1 *Le prédicteur de Bayes maximise la performance : $\forall h : \mathcal{X} \to Y, l_p(h) \geq l_p(h^*)$*

L'objectif est souvent de construire un classifieur pas « trop loin » de celui de Bayes, permettant ainsi d'étudier la performance de notre algorithme.

Théorème 24.1 *Soit $\mathcal{X} = [0,1]^d$, $\mathcal{Y} = \{0,1\}$ et $D = (X,Y)$ une distribution sur $\mathcal{X} \times \mathcal{Y}$ tel que la fonction de probabilité conditionnelle associée est c-lipschitzienne. Soit h_s le classifieur obtenue via l'algorithme 1-NN sur un ensemble de tirages $S \sim (X,Y)^n$. Alors,*

$$\mathbb{E}_{S\sim D^n}[l_P(h_s)] \leq 2l_P(h^*) + 4c\sqrt{d}n^{-\frac{1}{d+1}}$$

II Apprentissage non supervisé, clustering

Dans un problème d'apprentissage non supervisé, nos données sont simplement un ensemble de points $S = (x_1, ..., x_n) \in \mathcal{X}^n$ sans aucun étiquetage. Le but étant ici d'apprendre sur nos données, nous allons essayer de distinguer des similarités entre elles.
On considère donc le problème de classification (clustering) :
Entrées : n points $S = (x_1, ..., x_n) \in \mathcal{X}^n$.
Sortie : une partition $C_1, ...C_k$ de S.

Exemple 24.3 (Problème de clustering) *On dispose de plusieurs génomes, et on souhaite partitionner les individus en plusieurs catégories.*

On suppose que l'on dispose d'une distance d sur $\mathcal{X}$.

A Algorithme de classification hiérarchique ascendante

On initialise l'algorithme avec un cluster par point. À chaque itération, on calcul la distance minimal entre deux clusters et on les fusionne.

Algorithme 24.3 : Hiérarchie ascendante

pour $i = 1...n$ **faire**
  $c_i = \{x_i\}$;
pour $j = 1...n$ **faire**
  $(c_{i_j}, c_{k_j}) \leftarrow \text{argmin}_{c,c' \in \mathcal{C}} d(c,c')$;
  $\mathcal{C} \leftarrow \mathcal{C} \backslash \{c_{i_j}, c_{k_j}\} \cup \{c_{i_j} \cup c_{k_j}\}$

Pour calculer la distance entre deux clusters, on a plusieurs choix comme le minimum entre deux points du cluster, la distance moyenne ou la distance maximale.

B Algorithme des k-moyennes

Principe. On choisit k points au hasard pour commencer. L'algorithme fonctionne de manière itérative. Pour chaque point, on regarde de quel centre il est le plus proche et ensuite, on recalcule les centres des clusters, en prenant comme centroïde la moyenne de chaque cluster.

Fonction objectif. Pour évaluer la performance d'une partition $C_1, ..., C_k$, on pose $G_k(C_1, ..., C_k)$ la somme des distances quadratiques de chaque point à son centroïde correspondant.

Algorithme 24.4 : k-Moyennes

Données : $S \in \mathcal{X}^n, k \geq 0$
$\mu_1, ..., \mu_k \leftarrow \text{Random}(\mathcal{X})$;
tant que *non convergence* **faire**
  # Calcul des clusters **pour** $i = 1...k$ **faire**
    $C_i \leftarrow \{x \in S | i = \text{argmin}_j d(x, \mu_j)\}$;
  # Mise à jour des centroïdes **pour** $i = 1...k$ **faire**
    $\mu_i \leftarrow \frac{1}{|C_i|} \sum_{x \in C_i} x$;

Remarque 24.5 *Une version amélioré consiste à initialiser en prenant au hasard un point, puis un autre point loin premier, le troisième loin des deux premiers etc.*

Convergence. Le lemme suivant nous assure la terminaison mais ne nous dit rien sur la vitesse de convergence.

Lemme 24.1 *À chaque itération de l'algorithme des k-Moyennes, la fonction objectif diminue.*

Développement 17. Preuve du théorème ci-dessus.

Enfin, l'algorithme peut converger vers un point qui n'est même pas un minimum local de la fonction objectif.

C Minimisation du diamètre

On considère un problème similaire au précédent, le but étant cette fois de minimiser le maximum des diamètre d'un groupe. Étant donné $C \subset S$, on définit le diamètre de C par :

$$\delta(C) = \max_{x,x' \in C} d(x, x')$$

L'objectif est alors de trouver une partition $C_1, ..., C_k$ de S qui minimise

$$\max_{1 \leq i \leq k} \delta(C_i)$$

Ce problème étant $\mathbf{NP}$-complet, on pourra cependant trouver des algorithmes d'approximation.
Développement 18. Une 2-approximation au problème précédent.

Leçon 25

Analyses lexicale et syntaxique. Applications.

Auteur·e·s: Rousseau Guillaume, Marin Malory
Niveau : L3
Pré-requis : Théorie des langages, algorithmique
Références : [Legendre, 2015], [Sipser, 2013]

I Motivation

Compilation. Un compilateur est un programme qui transforme un code source en un code objet. Généralement, le code source est écrit dans un langage de programmation haut niveau facilement compréhensible par l'humain. Le code objet est généralement écrit en langage de plus bas niveau, par exemple un langage assembleur ou langage machine, afin de créer un programme exécutable par une machine.

Généralement, une chaîne de compilation se divise en trois partie :

1. le front-end (ou phase frontale), permettant de transformer le code source en pseudo-code ;
2. l'optimiseur de pseudo-code ;
3. le back-end (ou phase finale), permettant de transformer le pseudo-code en code objet.

On s'intéresse ici seulement au début de la phase frontale.

II Analyse lexicale

Principe. L'analyse lexicale est la conversion d'un texte en une liste de tokens (symboles) et elle fait partie de la première phase de la chaîne de compilation. Un programme réalisant une analyse lexicale est appelé un **analyseur lexical** (ou lexer).

Exemple. Un analyseur syntaxique peut recevoir en entrée la phrase ≪ Alfred Aho et Jeffrey Ullman ont reçu le prix Turing en 2020 ≫ , il retourne la liste des mots [Alfred, Aho, et, Jeffrey, Ullman, ont, reçu, le, prix, Turing, en, 2020].

A Lexèmes, expressions régulières et règle.

Lexèmes. Les mots produits par l'analyse lexicale sont appelés **lexèmes** (en anglais *tokens*). Pour reconnaître un lexème, on utilise pour cela les expressions régulières.

Expressions régulières. On fixe un alphabet Σ, qui est juste un ensemble de symboles, appelés **lettres**.

Définition 25.1 (Expression régulière et langage associé)
L'ensemble des expressions régulières se définit inductivement par :

1. $a \in \Sigma$, $\emptyset$ *et* ϵ *sont des expressions régulières ;*
2. *si* e_1 *et* e_2 *sont des expressions régulières, alors* $(e_1|e_2)$, $(e_2.e_2)$ *et* (e_1^*) *sont des expressions régulières.*

À chaque expression régulière, on associe un langage en prenant $\{a\}$, $\emptyset$ *et* $\{\epsilon\}$ *dans le cas 1, et en suivant les opérations dans le cas 2.*

Remarque 25.1 *On pourra confondre une expression rationnelle* e *et son langage associé* $L(e)$.

Exemple 25.1 *Étant donné l'alphabet* $\Sigma = \{0, 1\}$ *:*

1. $0^*10^* = \{w | w$ *contient un unique* $1\}$ *;*
2. $\Sigma^*001\Sigma^* = \{w | w$ *contient la sous-chaîne* $001\}$

Exemple 25.2 *L'ensemble des identifiants de la forme id.N où* `N` *est un nombre quelconque est le langage décrit par l'expression régulière* `id.(1|2|...|9)`*. *Ces expressions régulières sont très utiles pour décrire et trouver des chaînes vérifiant un pattern.*

Exercice 25.1 *Donner une expression régulière décrivant l'ensemble des nombres flottants.*

Règle. À chaque lexème, on ajoute un type. Par exemple, pour l'entrée « `k := 5 + 2*i` », le lexème `k` est du type `identifiant` et `5` du type `entier`.

Définition 25.2 (Règle de l'analyse lexicale) *Soit* Σ *un alphabet et* T *l'ensemble de type des lexèmes. Une* **règle d'analyse lexicale** *est la donnée :*

- *d'une expression régulière* e *sur* Σ*, dont le langage associé ,ne contient pas le mot vide* ϵ *;*
- *d'un type de lexème* $t \in T$.

On note cette règle $e \to t$.

B Automates finis

L'idée est maintenant de vérifier automatiquement si un mot est dans le langage d'une expression régulière. On utilise pour cela un automate fini.

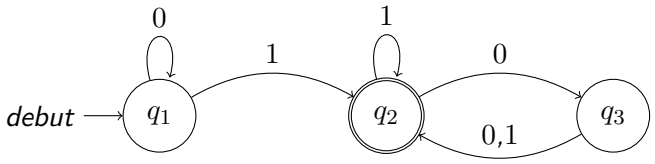
Définitions.

Définition 25.3 *Un automate fini est un quintuplet* $(Q, \Sigma, I, F, \delta)$ *où*

- Q *est un ensemble fini d'***états** *;*
- Σ *est un ensemble fini appelé* **alphabet** *;*
- $\delta : Q \times \Sigma \to Q$ *est la* **fonction de transition** *;*
- $q_0 \in Q$ *est l'***état initial** *;*
- $F \subset Q$ *est l'ensemble des* **états acceptants**.

Exemple 25.3 *Un automate* $M_1 = (Q, \Sigma, \delta, q_1, F)$ *où* $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{Q_2\}$, *et* δ *décrit par le tableau ci-dessous.*

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2



Étant donné un automate fini $M = (Q, \Sigma, \sigma, q_0, F)$ et $w = w_1...w_n$ un mot sur Σ, on dit que M accepte w s'il existe une suite d'états $r_0, ..., r_n$ vérifiant :

— $q_0 = r_0$;

— $\delta(r_i, w_{i+1}) = r_{i+1}$ pour $i = 0, ..., n-1$ et

— $r_n \in F$.

Étant donné un langage $L \subset \Sigma^*$, on dit que M reconnaît L si $L = \{w | M \text{ accepte } w\}$.

Exemple 25.4 *On revient à notre exemple M_1. Ici, on a*

$$L(M_1) = \{w \in \{0,1\}^* | w \text{ contient au moins un } 1 \text{ et un nombre pair de } 0 \text{ suive le dernier } 1\}$$

.

Expression régulière vers automate. On montre maintenant que pour toute expression régulière, son langage associé est reconnu par un automate finis.

Théorème 25.1 *Pour toute expression régulière e, il existe un automate fini M tel que $L(e) = L(M)$.*

Remarque 25.2 *Le sens réciproque de ce théorème est vrai : c'est le théorème de Kleene.*

Représentation d'un automate en machine. Il est très facile de simuler un automate sur un ordinateur. Il suffit de stocker la fonction de transition dans un tableau de dimension $|Q| \times |\Sigma|$.

C Algorithme d'analyse syntaxique

Principe. La méthode classique de décomposition d'un texte en lexèmes consiste à ordonnancer les règles et à reconnaître le plus long préfixe.

Soit $(e_i \to t_i)_{1 \leq i \leq n}$ un ensemble ordonné de règles et pour tout $1 \leq i \leq n$, soit A_i un automate reconnaissant $L(e_i)$.
On lit un caractère du texte :

— on fait lire ce caractère à chaque automate A_i ;

— si l'un des A_i est dans un état final, on a potentiellement trouvé un lexème du type t_i, on le retient ;

— on recommence jusqu'à ce que tous les automates soient bloqué.

— Parmi tous les lexèmes retenus, on choisit un de plus long préfixe et d'indice minimal en cas d'égalité ;

— si aucun lexème n'a été retenu, on lève une erreur.

Théorème 25.2 *Dans le pire des cas, l'analyse lexicale est quadratique en la taille du mot d'entrée.*

Exercice 25.2 *On considère les règle suivante $a \to$ lettre-a et $a^*b \to$ lettre-a-itérée-b. Compter le nombre de lectures pour analyser le texte a^n.*

D Applications

Coloration. Un analyseur lexical n'est pas utilisé qu'en compilation. Un éditeur de texte qui met des couleur à votre programme fait de la **coloration syntaxique** et utilise pour cela un analyseur lexical.

Erreurs. Ce même éditeur peut aussi détecter une erreur simple de syntaxe s'il ne détecte pas un lexème par exemple. De plus, en décorant un peu plus les lexèmes avec par exemple la ligne, sa valeur, etc, on peut afficher une erreur plus lisible pour le programmeur.

Remarque 25.3 *En cas d'erreur, notre logiciel peut calculer la distance d'édition du mot non reconnu avec certains lexèmes, afin de proposer une correction à l'utilisateur.*

III Analyse syntaxique

Principe. L'analyse syntaxique est la construction de l'**arbre syntaxique** représentant la structure d'une liste de lexèmes.

Exemple. L'analyse lexicale transforme « `k := 5+2*i` » en `[k,5,+,2,*,i]`, et l'analyse syntaxique transforme cette liste en l'arbre binaire correspondant.

A Grammaires hors-contexte

Les langages de programmations ont souvent une structure naturellement récursive. Par exemple, si b est un booléen, P_1 et P_2 deux programmes, alors `if` b `then` P_1 `else` P_2 est aussi un programme. Pour travailler sur la structure de tels langages, on introduit la notion de grammaire.

Définition 25.4 *Une* **grammaire hors contexte** *est un quadruplet (V, Σ, R, S) où :*

- *V est un ensemble fini appelé* **non terminaux** *;*
- *Σ est un ensemble fini, disjoint de V, appelé* **terminaux** *;*
- *R est un ensemble fini de* **règle**, *chacune étant une variable et un mot sur les variables et terminaux ;*
- *$S \in R$ est l'axiome.*

Dans le cas d'une analyse syntaxique, les terminaux correspondent aux lexèmes.

Exemple 25.5 *On considère la grammaire suivante :*

$$\begin{array}{lcl} S & \to & S+S \\ S & \to & c \end{array}$$

où le seul non-terminal est S, et le seul terminal est c.

Dérivation. On peut définir une dérivation d'un mot comme une application successive de règles. De manière plus formel, si u, v et w sont des mots contenant des terminaux et non-terminaux, et $A \to x$ est une règle, on dit que uAv « donne » uwv, et on note $uAv \Rightarrow uwv$.

Ainsi, on dit que v dérive de u, noté $u \Rightarrow^* v$, si $u = v$ ou il existe une suite $u_1, ..., u_k$ pour $k \geq 0$ avec

$$u \Rightarrow u_1 \Rightarrow u_2 ... \Rightarrow u_k \Rightarrow v$$

On appelle **longueur de dérivation** de u à v le nombre de règle appliquée, ici $k+1$.

Le **langage de la grammaire** est $\{w \in \Sigma^* | W \Rightarrow^* w\}$.

On parle de dérivation *leftmost* (resp. *rightmost*) lorsqu'à chaque règle appliquée, c'est le non-terminal le plus à gauche (resp. droite) qui est impliqué.

Ambiguïté d'une grammaire. Lorsque pour un même mot reconnu, il existe plusieurs dérivations distinctes, on dit que la grammaire est **ambiguë**.

B Éléments d'analyse syntaxique

Le premier but de l'analyse syntaxique est donc de déterminer si une phrase est reconnue par la grammaire ; autrement dit s'il existe une dérivation depuis l'axiome vers la phrase considérée.

L'objectif est alors double :

1. Décider l'appartenance au langage ;
2. connaître la structure (dérivation) de la phrase.

Analyse descendante. Un algorithme d'analyse descendante tente, à partir d'une liste de lexèmes, de construire une dérivation depuis l'axiome est de descendre jusqu'à la phrase.

Analyse ascendante. Un algorithme d'analyse ascendante tente, à partir d'une liste de lexèmes, de construire une dérivation à l'envers depuis la liste et de remonter jusque l'axiome.

C Une première solution : retour sur trace.

Une première idée consiste à faire de l'exploration exhaustive de l'ensemble des solutions. Cette solution est **générique** : elle fonctionne quelque soit la grammaire.

Idée de l'algorithme. Le but est d'explorer l'ensemble des dérivations jusqu'à obtenir un mot composé uniquement de terminaux, et le comparer au mot en entrée. Puisqu'il y a possiblement des branche infini, on utilise la borne suivante sur la taille des dérivations.

Théorème 25.3 *Soit G une grammaire et $m \in \Sigma^*$. Les deux assertions suivantes sont équivalentes :*

— $m \in L(G)$

— *il existe une dérivation reconnaissant m dont la longueur est inférieure à $a^{|m|\times r}$*

où a est le nombre maximum de symbole à droite d'une règle, et r le nombre de non-terminaux.

Algorithme 25.1 : Backtrack(G,u,u',l, $l_{\max}$)

Données : grammaire G, u mot d'entrée, u' mot en cours, l longueur de la dérivation en cours, $l_{\max}$ longueur maximal de dérivation
Sorties : VRAI ssi $u' \Rightarrow^* u$
si u' *contient uniquement des terminaux* **alors**
  retourner $u = u'$
sinon
  si $l > l_{\max}$ **alors**
    retourner *FAUX*
  sinon
    $N \leftarrow$ premier non terminal de u' ;
    pour *chaque règle* $N \to a_1 ... a_n$ **faire**
      $u'' \leftarrow u[N \mapsto a_1 ... a_n]$;
      si *Backtrack*(G,u,u'',$l+1$, $l_{\max}$) **alors**
        retourner *VRAI*
    retourner *FAUX*

Ainsi, il suffit d'exécuter Backtrack(G,u,S,0, $a^{|u|\times r}$) pour obtenir le résultat voulu.

La complexité exponentielle rend cet algorithme inutilisable en pratique, on va donc plutôt chercher d'autres méthodes.

D Analyse syntaxique en temps cubique

On présente ici un algorithme, appelé CYK (Cocke-Younger-Kasami), qui résout le problème du mot pour des grammaires sous forme normale de Chomsky.

Développement 11. Présentation du reste de l'algorithme CYK.

Malheureusement, une complexité cubique n'est toujours pas acceptable. On va pour cela utiliser des algorithmes gloutons non-générique.

IV Analyse syntaxique d'une grammaire LL(1)

L'analyse LL(1) est une analyse syntaxique descendante gloutonne en temps linéaire.

Signification de LL(1). *Left to right - Leftmost derivation* et le 1 signifie qu'on lit seulement le caractère courant pour trouver la dérivation.

A Définition de premier et suivant

Premier. Étant donné une grammaire G et un mot w composé de terminaux et non-terminaux, on définit un ensemble $\mathbf{premier}(w)$ correspondant à l'ensemble des lettres pouvant commencé un mot de terminaux u et tel que $v \Rightarrow^* u$.

Définition 25.5 *Soit G une grammaire, et $w \in (V \cup \Sigma)^*$, on définit :*

$$\mathbf{premier}(w) = \begin{cases} \mathbf{premier}'(w) & \text{si } w \not\Rightarrow^* \epsilon \\ \mathbf{premier}'(w) \cup \{\epsilon_0\} & \text{si } w \Rightarrow^* \epsilon \end{cases}$$

où :

$$\mathbf{premier}'(w) = \{a \in \Sigma | w \Rightarrow^* aw', w' \in (V \cup \Sigma)^*\}$$

Développement 12. Calcul de premier. On donne une définition axiomatique de $\mathbf{premier}$ et un algorithme de calcul par saturation.

Suivant. Pour tout non terminal N, on définit $\mathbf{suivant}(N)$ comme l'ensemble des terminaux pouvant apparaître après N dans une dérivation depuis S.

Définition 25.6 *Soit G une grammaire, et un non terminal N, on définit :*

$$\mathbf{suivant}(N) = \{a \in \Sigma | S \Rightarrow^* w_1 N a w_2, w_1, w_2 \in (V \cup \Sigma)^*\}$$

Calcul de suivant. De la même manière, on peut définir de manière axiomatique $\mathbf{suivant}$ et le calculer par saturation.

B Grammaire LL(1)

Définition 25.7 (Grammaire LL(1)) *Une grammaire G est dite LL(1) si, et seulement si, pour tout non-terminal N, en notant*

$$N \to w_1, ..., N \to w_n$$

l'ensemble des règles dont la partie gauche est N, on a :

- *les ensembles $\mathbf{premier}(w_i)$ pour $1 \leq i \leq n$ sont disjoints deux à deux ;*
- *si de plus $N \to \epsilon^*$, alors $\mathbf{suivant}(N)$ est disjoint de chaque $\mathbf{premier}(w_i)$.*

Exemple 25.6 *La grammaire $S \to +SS|c$ est LL(1), mais pas la grammaire $S \to SS + |c$ ne l'est pas.*

Ainsi, on en déduit un algorithme permettant de savoir si un mot m est engendré par une grammaire G qui est LL(1) linéaire en la taille du mot.

Leçon 26

Classes P et NP. Problèmes NP-complets. Exemples.

Auteur·e·s: Rousseau Guillaume
Niveau : MPI
Pré-requis : Notions d'algorithmique de graphes, de logique propositionnelle, de langages formels. La leçon ne mentionne pas les machines de Turing.
Références : [Carton and Perrin, 2008], [Benoit et al., 2013], [Arora and Barak, 2006], [Papadimitriou, 1994]

Introduction

L'objectif de cette leçon est de formaliser la notion de complexité, c'est à dire l'idée que certains problèmes sont plus durs que d'autres. Par exemple, il est facile d'écrire un programme qui, étant donné un graphe, *décide* si le graphe est connexe, et qui termine en un temps raisonnable. On ne connaît pas d'algorithme polynomial déterminant si un graphe est k-colorable, mais il est facile de *vérifier* si une k-coloration donnée est valide. Les notions de décision et de vérification sont au centre de l'étude des problèmes en informatique. Nous pouvons ainsi définir certaines classes de problèmes : ceux qui peuvent être décidés en un certain temps, ceux dont on peut vérifier une potentielle solution en un certain temps.

I Problèmes de décision

Définition 26.1 (Problème de décision) *Un* problème de décision *est un ensemble* d'instances E, *et un sous-ensemble* $P \subseteq E$ *d'instances dites* positives.

Voyons quelques exemples de problèmes

Exemple 26.1
— **Premier** *:* $E = \mathbb{N}$, $P = \{p | p\ premier\}$ *est le problème de savoir si un nombre donné est premier ou non.*
— **Connexe** *:* $E = \{G | G$ *graphe fini*$\}$, $P = \{G | G\ connexe\}$ *est le problème de savoir si un graphe donné est connexe ou non.*
— **Clique** *:* $E = \{(G,k) | G$ *graphe fini,* $k \in \mathbb{N}\}$, $P = \{(G,k) | G$ *graphe fini ayant une clique de taille* $k\}$ *est le problème de savoir si un graphe* G *contient une clique de taille* k.

Encodage Pour raisonner sur les problèmes avec un point de vue informatique, il faut refléter le fait que l'on travaille non pas sur des entiers, des graphes, mais sur des bits. Lorsqu'un problème traite d'objets complexes, il faut représenter, encoder ces objets sur des bits. Comme on peut facilement encoder n'importe quel alphabet fini sur des bits, on s'autorise à encoder des instances de problèmes sur un alphabet fini fixé Σ, qui peut être $\{0,1\}$, $\{0,1\}$ avec un symbole additionnel de séparation, etc... Un problème peut alors être vu comme le langage des mots qui correspondent au codage des instances positives.

Exemple 26.2 *Par exemple, un nombre entier n peut être encodé avec son écriture binaire, ou en unaire, c'est à dire avec une suite $\underbrace{1...1}_{n \text{ fois}}$.*
Un graphe G peut être encodé sur l'alphabet $\{0, 1, \bullet\}$ comme $n \bullet M$ où n est le nombre de sommets de G et M une suite de $n \times n$ booléens formant la matrice d'adjacence de G.

Définition 26.2 (Taille d'une instance) *On définit la* taille *d'une instance I comme sa longueur dans un encodage choisi. On la note $|I|$.*

Dans le cas des entiers, remarquons que le premier encodage prend une taille $\log(n)$, et le deuxième une taille n. Dans la suite, la métrique de performance des algorithmes permettant de résoudre un problème sera le nombre d'opérations élémentaires effectuées en fonction de la taille de l'instance. Le choix de l'encodage n'est donc pas trivial : pour les entiers, on a une différence d'échelle exponentielle sur la taille selon le codage !

Exercice 26.1 *Donner la taille de l'encodage par matrice d'adjacence d'un graphe à n sommets. Donner un encodage des graphes de taille $\log(n) + \log(m) + 2 * m * log(n)$.*

On considère donc dans la suite que les problèmes sont des langages sur l'alphabet Σ. Un programme répondant à un problème de décision est donc similaire à un automate fini : il prend en entrée un mot sur l'alphabet Σ et accepte ou rejette ce mot. La différence est qu'un automate fini a une expressivité limitée : le modèle mathématique d'une machine correspondant à la puissance de calcul des ordinateurs réels s'appelle la *Machine de Turing*.

II Complexité

La mesure de complexité d'un problème va être le temps que met un algorithme efficace à le résoudre.

Définition 26.3 (Complexité en temps) *Soit $f : \mathbb{N} \to \mathbb{N}$ une fonction. La classe de problèmes $\mathbf{DTIME(f(n))}$ est l'ensemble des problèmes P tels qu'il existe un programme décidant P en temps $O(f(n))$.*

A Classe P

Définition 26.4 *La classe $\mathbf{P}$ est la classe des problèmes décidables en temps polynomial :*

$$\mathbf{P} = \bigcup_{k=0}^{+\infty} \mathbf{DTIME}(n^k)$$

La classe $\mathbf{P}$ représente les problèmes considérés comme faciles en informatique, car décidables avec un programme qui peut tourner en pratique sur un ordinateur. En théorie cette classe contient des problèmes qui sont décidables avec des algorithmes en $O(n^{1000})$, et donc irréalisables, mais la plupart des problèmes réels rencontrés étant dans $\mathbf{P}$ sont décidables en $O(n^5)$ au maximum.

Exemple 26.3 *Les problèmes* **Premier** *et* **Connexe** *sont dans $\mathbf{P}$.*

Il existe certains problèmes dont on ne sait pas s'ils sont dans $\mathbf{P}$ ou pas. L'exemple canonique d'un tel problème est **SAT** : « Étant donné une formule booléenne φ sous forme normale conjonctive, existe-t-il une valuation σ telle que $\sigma(\varphi) = 1$ »

$2-\mathbf{SAT}$. On restreint le problème précédent aux formules booléennes telle que chaque clause ne contient que 2 littéraux : on appelle ce problème $2-\mathbf{SAT}$.

Théorème 26.1 *$2-\mathbf{SAT}$ est décidable en temps linéaire.*

Développement 14. Preuve du théorème ci-dessus.

B Certificat

Bien qu'on ne sache pas résoudre $\mathbf{SAT}$ en temps polynomial, on peut vérifier une solution à une instance de $\mathbf{SAT}$ en temps polynomial : étant donné une valuation σ, il est facile de déterminer $\sigma(\varphi)$. On souhaite formaliser cette notion de vérification de solution.

Définition 26.5 (NP) *Un problème L sur l'alphabet Σ est dans $\mathbf{NP}$ s'il existe un polynôme $p(n)$ et un problème $L' \in \mathbf{P}$ sur un alphabet Σ' contenant Σ tel que pour tout $x \in \Sigma^*$:*

$$x \in L \iff \exists u \in \Sigma'^{<p(|x|)}, xu \in L'$$

Lorsque $x \in L$ et u est tel que $xu \in L'$, on dit que u est un certificat *pour x.*

En d'autres termes, un problème est dans $\mathbf{NP}$ s'il existe un protocole polynomial permettant de décider une instance si l'on lui donne un indice, sous la forme du certificat. Notons que tout problème dans $\mathbf{P}$ est aussi dans $\mathbf{NP}$: il suffit de prendre $L' = L$ dans la définition précédente. En revanche, personne n'a pu prouver ou infirmer que $\mathbf{P} = \mathbf{NP}$, c'est même un des problèmes du prix du millénaire. Il se pourrait même que cette question soit indécidable dans le modèle mathématique actuel !

Exemple 26.4 *Souvent, le certificat prend la forme d'une solution au problème. Considérons le problème $\mathbf{SAT}$ défini plus haut. Un bon candidat pour un certificat d'une instance positive φ est une valuation σ telle que $\sigma(\varphi) = 1$. Un tel certificat est bien encodable en taille polynomiale en $|\varphi|$, et donc $\mathbf{SAT}$ est bien dans $\mathbf{NP}$.*

III NP-complétude

Nous avons donc vu qu'il existe des problèmes faciles à vérifier, les problèmes $\mathbf{NP}$. On veut s'intéresser aux problèmes les plus durs de la classe $\mathbf{NP}$. En effet, nous allons voir que certains problèmes sont $\mathbf{NP}$-complets, ce qui signifie que s'ils sont dans P, alors $\mathbf{P} = \mathbf{NP}$. Autrement dit, ils sont au moins aussi durs que tous les problèmes $\mathbf{NP}$.

A Réduction polynomiale

Définition 26.6 (Réduction polynomiale, NP-complétude) *Soient A et B deux problèmes. On dit que A est* réductible en temps polynomial au sens de Karp *à B, noté $A \leq_p B$, s'il existe un programme F polynomial transformant des instances de A en des instances de B et tel que pour toute instance I de A, $I \in A \iff F(I) \in B$.*

On dit que B est $\mathbf{NP}$-dur si tout problème A de $\mathbf{NP}$ s'y réduit, on dit que B est $\mathbf{NP}$-complet si de plus B est dans $\mathbf{NP}$.

Remarquons que la relation $\leq_p$ est transitive (exercice : le montrer. Indice : une composition de polynôme est un polynôme), et donc si un problème B est dans $\mathbf{P}$ et se réduit à un problème B', alors B' est aussi dans $\mathbf{P}$.
Supposons maintenant que B est $\mathbf{NP}$-complet et se réduit à un problème B'. Alors B' est aussi $\mathbf{NP}$-complet. Le problème est donc d'exhiber un premier problème $\mathbf{NP}$-complet, à partir duquel faire des réductions.

Théorème 26.2 (Théorème de Cook, Admis) $\mathbf{SAT}$ *est* $\mathbf{NP}$*-complet.*

Cette preuve utilise les Machines de Turing, l'idée étant d'encoder le comportement d'une telle machine dans une formule booléenne.

B Exemples de problèmes NP-complet

Donnons maintenant quelques exemples de problèmes $\mathbf{NP}$-complets.

Définition 26.7 (3-SAT) *Étant donné une formule booléenne ϕ sous forme normale conjonctive telle que chaque clause contienne 3 littéraux, existe-t-il une valuation v telle que $v(\phi)$ soit vrai ?*

Définition 26.8 (CLIQUE, VERTEX − COVER) *Étant donné un graphe G et un entier K :*

- $\mathbf{CLIQUE}$ *: G admet-t-il un sous-graphe complet à K sommets ?*
- $\mathbf{VERTEX - COVER}$ *: existe-t-il un sous-ensemble de K sommets ou moins tels que chaque arête soit adjacente à au moins un de ces sommets ?*

Définition 26.9 (2-PARTITION) *Étant donné n entiers $a_1, ..., a_n$, existe-t-il un sous-ensemble $I \subset [n]$ tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.*

Théorème 26.3 *3-*$\mathbf{SAT}, \mathbf{CLIQUE}, \mathbf{VERTEX - COVER}$ *et 2-*$\mathbf{PARTITION}$ *sont* $\mathbf{NP}$*-complets.*

Lorsqu'on étudie un problème d'optimisation, il n'est pas toujours concevable de trouver une solution optimale en temps polynomiale. On utilisera alors des **algorithmes d'approximations** qui permettent de donner une solution ≪ proche ≫ de l'optimale tout en s'exécutant en temps polynomiale.

C Définitions

On s'intéresse ici exclusivement à des problèmes d'optimisations, dont on rappelle la définition.

Définition 26.10 (Problème d'optimisation) *Un problème d'optimisation est un problème $\mathcal{P} = (I, S)$ muni d'une* **fonction d'évaluation** *$c : I \times S \to \mathbb{R}^+$ calculable en temps polynomial et d'une* **fonction objectif** *$o \in \{\min, \max\}$.*

Étant donné une instance $i \in I$ de $\mathcal{P}$, l'objectif du problème d'optimisation $\mathcal{P}$ est de construire une solution $s^* \in S(i)$ vérifiant :

$$c(i, s^*) = o\{c(x, s) : s \in S(i)\}$$

Définition 26.11 (Algorithme d'approximation) *Une λ-approximation pour un problème d'optimisation $\mathcal{P}$ est un algorithme A ayant un temps d'exécution polynomial en la taille de l'instance et qui retourne une solution approximée qui est dans le pire des cas, à un facteur λ de la solution optimale. Autrement dit, pour toute instance $i \in I$, $A(i) \in S(i)$ et*

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \lambda$$

où $C = c(i, A(i))$ et $C^* = c(i, s^*)$ avec s^* une solution optimale pour l'instance i.

D Exemples

Couverture de sommets (Vertex cover). Le problème $\mathbf{VERTEX - COVER}$ défini précédemment peut être réécrit sous la forme d'un problème d'optimisation : le but est alors de trouver une couverture de sommet minimum.

Algorithme 26.1 : Glouton-VC

Données : Un graphe $G = (V, E)$.
Résultat : Un couverture des sommets de G, noté S.
$S \leftarrow \emptyset$;
tant que $\exists (u, v) \in E, u, v \notin S$ **faire**
  choisir $(u, v) \in E$ telle que $u, v \notin S$
  $S \leftarrow S \cup \{u, v\}$

Théorème 26.4 *L'algorithme Glouton-VC est une 2-approximation pour le problème de couverture de sommet.*

E Non-approximabilité

De la même manière que l'on peut montrer que des problèmes sont trop durs pour être résolu par des algorithmes en temps polynomiale, on peut montrer que certain problème ne peuvent même pas être approximer. L'exemple le plus connu est le problème du **voyageur de commerce**.

Problème du voyageur de commerce.

Théorème 26.5 *Pour tout $\lambda \geq 1$, il n'existe aucune λ-approximation pour le problème du voyageur de commerce à moins que $\mathbf{P} = \mathbf{NP}$.*

Remarque 26.1 *La méthode général pour prouver qu'il n'existe pas d'algorithme d'approximation est souvent le même que pour le théorème précédent : on suppose qu'un tel algorithme existe et on construit un algorithme polynomial qui résout un problème NP-complet.*

Exercice 26.2 *Montrer que si on suppose que la fonction de coût satisfait l'inégalité triangulaire , on peut trouver une 2-approximation du problème du voyageur de commerce.*

Développement 13. Étude complète du problème du voyageur de commerce.

F Algorithme d'approximation probabiliste pour satisfaisabilité MAX-3-CNF

Il est possible d'étendre la notion d'algorithme d'approximation aux algorithmes probabilistes en considérant C comme l'espérance du coût.

Définition 26.12 (MAX-3-CNF) *Étant donné n variables $x_1, x_2, ..., x_n$ et une formule sous forme normale conjonctive à m clauses contenant chacune 3 littéraux, maximiser le nombre de clauses satisfaite avec une assignation.*

Théorème 26.6 *Soit une instance de MAX-3-CNF à n variables $x_1, x_2, ..., x_n$ et m clauses ; alors, l'algorithme randomisé qui affecte indépendamment à chaque variable la valeur 1 avec une probabilité $1/2$ et la valeur 0 sinon est une $8/7$-approximation randomisée.*

Leçon 27

Décidabilité et indécidabilité. Exemples.

Auteur·e·s: Bertrand Jules
Niveau : L3
Pré-requis : Automates, Langages
Références : [Sipser, 2013], [Carton and Perrin, 2008]

On présente ici un nouveau modèle de calcul, proposé par Alan Turing en 1936, appelé **machine de Turing**. Ces machines sont similaires à un automate finis, mais avec une quantité illimité de mémoire. Ainsi, une machine de Turing peut autant faire que n'importe quel ordinateur, et pourtant il existe certains problème qu'elles ne peuvent pas résoudre.

I Machine de Turing et décidabilité

A Modèle de calcul

Définition 27.1 *Une* **machine de Turing** *est un septuplet* $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ *où :*

1. Q *est l'ensemble des états ;*
2. Σ *est l'alphabet d'entrée, ne contenant pas le symbole* B *(blanc) ;*
3. Γ *est l'alphabet de travail, avec* $B \in \Gamma$ *et* $\Sigma \subset \Gamma$ *;*
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *est la fonction de transition ;*
5. $q_0 \in Q$ *est l'état initial ;*
6. $q_a \in Q$ *est l'état acceptant ;*
7. $q_r \in Q$ *est l'état rejetant, avec* $q_r \neq q_a$.

Configuration. Lorsqu'une machine calcule, elle change d'état, de contenue du ruban ainsi que la position de la tête. Ces trois éléments forment la **configuration** de la machine. De manière générale, on représente une configuration par un triplet uqv où q est l'état courant, uv est la chaîne sur le ruban et la tête est positionné sur le premier caractère de v. On appelle **configuration initiale** (resp. acceptante / rejetante) toute configuration dont l'état est initiale (resp. acceptant/rejettant).

Calcul. On peut alors formaliser la notion de calcul. On appelle pas de calcul le passage d'une configuration C_1 à une configuration C_2. Un **calcul** est une suite de pas de calcul partant d'une configuration initiale à une configuration finale.

Remarque 27.1 *On peut définir les machines de Turing avec de nombreuses variations :*

— *possibilité pour la tête de lecture de rester sur place ;*

— *utilisation de multiples rubans infini ou bi-infini ;*

> — *restriction de Σ à $|\Sigma| = 2$;*
> — *non déterminisme dans les transitions.*
>
> *Ces variantes restent équivalentes au modèle proposé.*

B Langage reconnu et décidable

Étant donné une machine de Turing M et un mot w, on dit que M **accepte** (resp. **rejette**) w s'il existe un calcul acceptant de M sur l'entrée w.

Langage accepté, décidable. Après avoir définit une notion de calcul, on peut maintenant s'intéresser à la décidabilité.

Définition 27.2 (Mot et langage accepté) *L'ensemble des mots acceptés par une machine de Turing M, noté $L(M)$, est appelé* **langage accepté** *par M.*
Un langage L est **reconnaissable** *par machine de Turing s'il existe une machine M telle que $L(M) = L$.*

Lorsqu'on lance une machine de Turing, il y a trois possibilités : le mot est accepté, rejeté ou la machine boucle (ne termine pas). Lorsqu'une machine ne boucle jamais, on dit qu'elle décide $L(M)$: elle accepte w si $w \in L(M)$ et rejette sinon.

Définition 27.3 (Langage décidable) *Un langage L est dit* **décidable** *s'il existe une machine M qui décide L.*

Exemple de machine de Turing. Les machines de Turing peuvent être représentées via un graphe orienté de la même manière que les automates finis. On peut aussi, de manière moins formelle, décrire le fonctionnement de la machine. Considérons le langage $L = \{0^{2^n} | n \geq 0\}$ sur l'alphabet $\Sigma = \{0\}$. La machine M décrite ci-dessous décide L :

$\mathbf{M}$ = " Sur l'entrée w :
1. Traverser le ruban de gauche à droite en rayant un 0 sur deux.
2. Si, à l'étape 1, le ruban contient un unique 0, **accepter**.
3. Si, à l'étape 1, le ruban contient plus qu'un seul 0 et que ce nombre est impair, **rejeter**.
4. Remettre la tête de lecture sur le bord gauche du ruban.
5. Revenir à l'étape 1.

Exercice 27.1 *Représenter la machine M ci-dessus sur un graphe orienté.*

Des problèmes décidables. On rappelle qu'un **problème** est la donnée de la représentation des éléments d'en ensemble au plus dénombrable et d'une question sur ces éléments. Par exemple, le problème qui consiste à déterminer si un entier en base 10 est premier n'est pas le même lorsque les entiers sont donnés via leur décomposition en facteurs premiers. Un problème $\mathcal{P} = (I, Q)$ (où I est l'ensemble des instances et Q une proposition sur I) peut se voir comme le langage $L(\mathcal{P}) = \{\langle x \rangle \in I | Q(x)\}$. On dira alors que le problème $\mathcal{P}$ est décidable lorsque $L(\mathcal{P})$ l'est. On pourra alors confondre problème et langage dans les sections suivantes.

Problème indécidable. Par argument diagonale, on peut montrer l'existence d'un problème indécidable.

Proposition 27.1 *Il existe des problèmes indécidables.*

Remarque 27.2 *Cette preuve peut aussi se faire par un argument de dénombrement. En effet, il existe un nombre indénombrable de langages alors qu'il y a un nombre dénombrable de machines de Turing.*

C Robustesse du modèle et thèse de Church

D'autres modèles de calculs. Les machines de Turing ne sont pas les seuls modèles de calcul existants.

Définition 27.4 *Les* **fonctions récursives** *sont l'ensemble des fonctions :*

- *contenant les fonctions constantes entières, les projections et la fonction successeur ;*
- *clos par composition, récurrence et minimisation.*

Définition 27.5 *Les* **machines RAM** *sont composées de :*

- *deux bandes infinies, une d'entrée et une de sortie ;*
- *des registres en nombre arbitrairement grand ;*
- *un programme composé d'opérations assembleur.*

Remarque 27.3 *Les machines RAM peuvent être vues comme des ordinateurs à mémoire infinie ; tandis que les fonctions récursives comme des fonctions écrites dans un langage fonctionnel.*

Tous ces modèles définissent une notion de calcul différente de celles proposée par les machine de Turing, mais ne change pas la notion de décidabilité. Autrement dit, un langage est décidable par machine de Turing si, et seulement si, il est calculable par une fonction récursive.

Ces remarques ont mené à la **thèse de Church** : la notion de calculabilité définie par les machines de Turing correspond exactement à la notion de calcul naturel. Autrement dit, tout ce qui est calculable par un système physique est calculable par une machine de Turing.

Remarque 27.4 *Cette thèse n'est pas démontrable dans le cadre de l'informatique. Elle nécessite d'aborder la notion de « calcul naturel » ou de « système physique ».*

II Exemples de problèmes décidables et indécidable

A Preuve de décidabilité et d'indécidabilité

Pour prouver qu'un problème est décidable, il suffit d'exhiber une instance d'un modèle de calcul qui le résout.

Exemple 27.1 *Le langage $\mathcal{B} = \{w\#w | w \in \{0,1\}^*\}$ est décidable. On peut le montrer avec trois niveaux de granularité de preuve :*

1. *en exhibant une machine M tel que $L(M) = \mathcal{B}$;*
2. *en décrivant le fonctionnement d'une machine M tel que $L(M) = \mathcal{B}$;*
3. *en donnant un algorithme qui décide $\mathcal{B}$.*

Sinon, on peut transformer un problème en un autre problème que l'on sait déjà décidable. De la même manière, on peut montrer qu'un problème est indécidable en le réduisant à un problème déjà connu.

Fonction calculable. On dira qu'une fonction $f : \Sigma^* \to \Sigma^*$ est **calculable** s'il existe une machine de Turing M qui, sur chaque entrée w, s'arrête avec seulement $f(w)$ sur son ruban. Par exemple, toutes les fonctions arithmétiques usuels sont calculables.

Réduction. Les fonctions calculables peuvent alors être des transformations de code de machines.

Définition 27.6 *Soient A et B deux langages d'alphabets respectifs Σ_A et Σ_B. Une réduction de A à B est une fonction calculable $f : \Sigma_A^* \to \Sigma_B^*$ telle que*

$$w \in A \Leftrightarrow f(w) \in B$$

On note alors $A \leq_m B$ lorsque A se réduit à B.

Proposition 27.2 *Soient A et B tels que $A \leq_m B$.*

— *Si B est décidable, alors A est décidable.*

— *Si A est indécidable, alors B est indécidable.*

B indécidabilité de problèmes liés aux machine de Turing

Code des machines. Pour une machine de Turing M, et pour tout mot w, on note $\langle M, w\rangle$ le codage du couple (M, w).

On définit le **langage d'acceptation** est $L_\in = \{\langle M, w\rangle | w \in L(M)\}$.

Machine universelle. On appelle **machine universelle** une machine de Turing pouvant simuler n'importe quelle machine de Turing sur n'importe quelle entrée.

Proposition 27.3 *Il existe une machine universelle $\mathbf{U}$. De plus, on a $L(\mathbf{U}) = L_\in$.*

Théorème 27.1 *$L_\in$ est indécidable.*

Problème de l'arrêt. On définit un problème plus intéressant en informatique : le **problème de l'arrêt**. On considère le langage $L_A = \{\langle M, w\rangle | M$ s'arrête sur l'entrée $w\}$.

Proposition 27.4 *L_A est indécidable.*

C Décidabilité et théorie des langages

Langages rationnels. On peut montrer assez facilement que les langages rationnels sont décidables. Les problèmes liés à ces langages peuvent cependant déjà être indécidables.

Exercice 27.2 *Montrer que tout langage rationnel est décidable.*

Développement 28. Décidabilité et langages rationnels.

Problème de correspondance de Post. Le problème de correspondance de Post est un problème de décision indécidable introduit par Emil Post en 1946. Comme il est plus simple que le problème de l'arrêt et de l'acceptation, il apparaît souvent dans des démonstrations d'indécidabilité.

Définition 27.7 (POST) *Étant donné une séquence de couple de mots $(a_1, b_1), ..., (a_n, b_n)$ sur un alphabet Σ, existe-t-il un mot $w \in \Sigma^*$ tel qu'il existe une suite d'indice $i_1, ..., i_k$ vérifiant $w = a_{i_1}...a_{i_k} = b_{i_1}...b_{i_k}$.*

Théorème 27.2 $\mathbf{POST}$ *est indécidable.*

Ce problème est très utile pour montrer des résultats sur les grammaires et langages algébriques ;

Théorème 27.3 *Les problèmes suivants sont indécidables :*

- *Étant donné une grammaire algébrique G, est-elle ambiguë ?*
- *Étant donné une grammaire algébrique G, son langage est-il Σ^* ?*
- *Étant donné des grammaires algébriques G_1 et G_2, leurs langages sont-ils égaux ?*

D Problème de logique

La théorie de la calculabilité trouve une application importante en logique mathématiques. On dira qu'une théorie logique est décidable si l'ensemble des formules closes et vraies est décidable.

Arithmétique de Presburger. On définit l'arithmétique de Presburger comme la théorie au premier ordre des entiers munis de l'addition.

Théorème 27.4 *L'arithmétique de Presburger est décidable.*

Arithmétique de Peano. L'arithmétique de Peano est la théorie au premier ordre des entiers munis de l'addition et de la multiplication.

Théorème 27.5 (admis) *L'arithmétique de Peano est indécidable.*

III Langages récursivement énumérable

A Définitions et exemples

Un autre terme pour les langages reconnu par une machine de Turing est « récursivement énumérable ». Ce terme vient d'une variante des machines de Turing, appelée **énumérateur**. De manière informelle, un énumérateur est une machine de Turing auquel on ajoute une imprimante avec laquelle une machine de Turing peut sortir un mot. L'ensemble des mots « imprimés » par un énumérateur est appelé **langage énuméré**.

Exercice 27.3 *Proposer une définition formelle d'un énumérateur.*

Théorème 27.6 *Un langage est reconnu par une machine de Turing si, et seulement si, il est énuméré par un énumérateur.*

Un langage est alors dit **co-récursivement énumérable** si son complémentaire est récursivement énumérable.

Exercice 27.4 *Montrer qu'un langage est décidable si, et seulement si, il est récursivement énumérable et co-récursivement énumérable.*

B Théorème de Rice

Cette notion permet d'introduire un théorème très puissant sur l'indécidabilité de certains langages : le **théorème de Rice**. De manière informelle, il peut s'interpréter comme l'indécidabilité de la correction des programmes.

Développement 29. Théorème de Rice et applications.

Leçon 28

Formules du calcul propositionnel : représentation, formes normales, satisfiabilité. Applications.

Auteur·e·s: Marin Malory
Niveau : début d'un cours de logique niveau L3
Pré-requis : Définition par induction, logique mathématiques basique
Références : [Pinchinat et al., 2022]

I Formules

A Syntaxe de la logique propositionnelle

Le langage de la logique propositionnelle repose sur la notion de variable propositionnelle, que l'on note souvent p, q, x, y, z et qui est juste un ensemble de symbole que l'on notera X.

Définition 28.1 *On définit les* **formules de la logique propositionnelle** *inductivement :*

- *toute variable propositionnelle $x \in X$ est une formule propositionnelle ;*
- *$\top$ et $\bot$ sont des formules ;*
- *si φ est une formule, alors $\neg\varphi$ en est une ;*
- *si φ et ψ sont des formules, alors $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ et $(\varphi \to \psi)$ sont des formules.*

On appelle **connecteur logique** les symboles permettant de connecter les formules, comme $\neg$, $\wedge$, $\vee$ et $\to$.

Exemple 28.1 *Pour $X = \{p, q, r\}$, les mots $(p \to (\neg q \wedge r))$ et $((p \to \neg q) \wedge r$ sont des formules.*

Arbre de syntaxe. Via la définition récursive, une formule peut être représenté par un arbre. Cette bijection entre les arbres syntaxiques et les formules est justifiée par le théorème de lecture unique.

Théorème 28.1 (Théorème de lecture unique) *Soit ϕ une formule. Un seul des cas suivant est possible :*

- *ϕ est une variable propositionnelle ;*
- *il existe une unique constante $c \in \{\top, \bot\}$ telle que $\varphi = c$;*
- *il existe une unique formule ψ telle que $\varphi = \neg\psi$;*

— *il existe une unique de formule* (ψ_1, ψ_2) *et un unique opérateur* $o \in \{\wedge, \vee, \to\}$ *tels que* $\varphi = (\psi_1\ o\ \psi_2)$.

Taille, hauteur et sous-formule. La taille $|\varphi|$ et la hauteur $h(\varphi)$ d'une formule sφ e définisse comme la taille et la hauteur de l'arbre syntaxique unique associée. Une sous-formule $\mathbf{SF}(\varphi)$ est la formule associée à un sous-arbre de l'arbre syntaxique.

Exemple 28.2 *Pour* $\varphi = (p \to (\neg q \wedge r))$*, on a* $|\phi| = 6$*,* $h(\varphi) = 3$ *et* $\mathbf{SF}(\varphi) = \{\varphi, p, q, r, (\neg q \wedge r), \neg q\}$.

B Sémantique de la logique propositionnelle

Valuation. Étant donné un ensemble de variable propositionnelle X, une valuation sur X est une fonction $v : X \to \{0, 1\}$.

Valeur de vérité d'une formule. On peut étendre le domaine de définition d'une valuation au formule et ainsi définir la valeur de vérité de cette formule.

Définition 28.2 *Soit* φ *une formule sur un ensemble de variables* X *et* $v : X \to \{0, 1\}$ *une valuation. On définit* **l'évaluation de** φ **pour** v *par :*

— $\overline{v}(x) = v(x)$ *pour* $x \in X$;
— $\overline{v}(\top) = 1$ *et* $\overline{v}(\bot) = 0$;
— $\overline{v}(\neg\psi) = 1$ *ssi* $\overline{v}(\psi) = 0$;
— $\overline{v}(\psi_1 \wedge \psi_2) = 1$ *ssi* $\overline{v}(\psi_1) = 1$ *et* $\overline{v}(\psi_2) = 1$;
— $\overline{v}(\psi_1 \vee \psi_2) = 1$ *ssi* $\overline{v}(\psi_1) = 1$ *ou* $\overline{v}(\psi_2) = 1$; $\overline{v}(\psi_1 \to \psi_2) = 1$ *ssi* $\overline{v}(\psi_1) = 1$ *implique* $\overline{v}(\psi_2) = 1$;

Si $\overline{v}(\varphi) = 1$*, on note* $v \models \varphi$ *: «* v *satisfait* φ *».*

Exemple 28.3 *On reprend l'exemple précédent avec* $v(p) = 1 = v(r) = 1$ *et* $v(q) = 0$.

Table de vérités. Une représentation de la sémantique d'une formule est possible par table de vérité. Dans une telle table, chaque ligne correspond à une valuation et la dernière colonne à la valeur de vérité pour cette valuation.

Exercice 28.1 *Dresser la table de vérité pour le formule de l'exemple précédent.*

Satisfiabilité,tautologie.

Définition 28.3
Une formule φ *est* **satisfiable** *s'il existe une valuation* v *telle que* $v \models \varphi$.
Une formule φ *est une* **tautologie** *(ou est* **valide***) si pour toute valuation* v*, on a* $v \models \varphi$*. On note alors* $\models \varphi$.
Une formule φ *est un* **contradictoire** *(ou est* **insatisfiable***) si pour toute valuation* v*, on a* $v \not\models \varphi$.

Proposition 28.1 *La formule* φ *est une tautologie si, et seulement si,* $\neg\varphi$ *est contradictoire.*

Remarque 28.1 *Pour toute formule* ϕ*, on a* $\models \varphi \vee \neg\varphi$ *(tiers-exclu) et* $\models \neg(\varphi \wedge \neg\varphi)$ *(non contradiction).*

C Équivalence et conséquence

Équivalence. La sémantique des formules nous permet de définir une notion d'équivalence sur les formules.

Définition 28.4 (Équivalence) *Deux formules φ et ψ sont dit* **sémantiquement équivalente** *si pour toute valuation v, on a $v \models \varphi$ ssi $v \models \psi$. On note alors $\varphi \equiv \psi$.*

Exercice 28.2 *Montrer que $\equiv$ définit une relation d'équivalence.*

On peut présenter quelques équivalences classiques :

— élément neutre : $\varphi \wedge \top \equiv \varphi$, $\varphi \vee \bot \equiv \varphi$

— associativité et commutativité de $\wedge$ et $\vee$

— distributivité

— loi de De Morgan

— négation

Proposition 28.2 *Soit φ et ψ deux formules. On a $varphi \equiv \psi$ ssi $\models \varphi \leftrightarrow \psi$.*

Conséquence logique. Cette relation nous permet des sortes d'affaiblissement.

Définition 28.5 *Soit φ et ψ deux formules. On dit que ψ est conséquence logique de φ, noté $\varphi \models \psi$ si pour toute valuation v, si $v(\phi) = 1$ alors $v(\psi) = 1$.*

Remarque 28.2 *On a alors $\varphi \equiv \psi$ ssi $\varphi \models \psi$ et $\psi \models \varphi$.*

II Fragments syntaxiques

A Système complet de connecteur

On parle de **système de connecteurs** pour parler d'un ensemble de connecteur logiques, comme $\{\neg, \wedge, \vee, \rightarrow\}$.

Définition 28.6 *Un système de connecteurs est* **complet** *si toute formule est équivalente à une formule ne s'écrivant qu'avec les connecteurs de ce système.*

Proposition 28.3 *$\{\neg, \wedge\}$ est un système complet de connecteur.*

On définit `NAND` comme la négation d'une conjonction.

Proposition 28.4 *Le système* $\{\texttt{NAND}\}$ *est complet.*

Application aux circuits booléens. Un circuit logique peut être défini comme un graphe orienté acyclique, ayant des noeuds spéciaux appelés entrées et sorties et dont les noeuds intermédiaires sont des connecteurs logiques. Le théorème précédent nous affirme que tout circuit peut être réalisé seulement avec des portes NAND.

B Formes normales

Définition 28.7 *On appelle* **littéral** *toute variable propositionnelle ou négation de variable propositionnelle.*

Forme normale conjonctive. On définit une **clause** comme une disjonction de littéraux. Ainsi, une formule sous forme normale conjonctive est une conjonction de clauses.

Définition 28.8 *Une formule* φ *est sous forme normale conjonctive s'il existe des littéraux* l_{ij} *pour* $1 \leq i \leq n$ *et* $1 \leq j \leq m$ *tels que :*

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} l_{ij}$$

Proposition 28.5 *Toute formule* φ *est équivalente à une autre formule sous forme normale conjonctive.*

La preuve de la proposition précédente nous donne un algorithme permettant de mettre une formule sous forme normale conjonctive, l'idée étant pour chaque sous-formule, d'envoyer vers l'avant les $\wedge$.

Forme normal disjonctive. Il existe une autre forme normale, duale à la première.

Définition 28.9 *Une formule* φ *est sous forme normale conjonctive s'il existe des littéraux* l_{ij} *pour* $1 \leq i \leq n$ *et* $1 \leq j \leq m$ *tels que :*

$$\varphi = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} l_{ij}$$

Proposition 28.6 *Toute formule* φ *est équivalente à une autre formule sous forme normale disjonctive.*

La preuve de la proposition précédente nous donne un algorithme permettant de mettre une formule sous forme normale conjonctive, l'idée étant de dresser la table de vérité, puis d'en déduire directement la formule.

Exercice 28.3 *Donner un moyen effectif de passer d'une forme normale à une autre.*

Exercice 28.4 *Combien existe-t-il de formules à équivalence près ?*

III Problème SAT

Même si le pouvoir d'expression de la logique propositionnelle semble limité, il permet notamment de modéliser un bon nombre de problème. On discute ici de la notion de satisfiabilité, problème basique en théorie de la complexité.

A Problème SAT, 3-SAT et complexité

Coloriage de graphe. Il existe un problème très classique en théorie des graphes, qui est celui du **coloriage**. Étant donné un graphe $G = (V, E)$et un entier $k > 0$, existe-t-il une fonction de coloriage $c : V \to \{1, ..., k\}$ tel que pour toute arête $ij \in E$, $c(i) \neq c(j)$.

Exercice 28.5 *Montrer qu'il existe une formule $\phi_{G,k}$ telle que ϕ_G, est satisfiable si, et seulement si, G est k-coloriable.*

On se ramène alors à un problème de satisfiabilité d'une formule. On peut faire de même avec de nombreux problèmes ou jeu, comme le sudoku par exemple.

Problème SAT.

Définition 28.10 **SAT** *Étant donné une formule φ sous forme normale conjonctive, φ est-elle satisfiable ?*

Remarque 28.3 *On peut supposer φ qui n'est pas sous forme normale, cela ne change pas les énoncés suivants.*

Décidabilité et complexité. Quelques résultats viennent de la théorie de la complexité, et nous permette de mieux comprendre le problème **SAT**.

Proposition 28.7 *Le problème* **SAT** *est décidable.*

Théorème 28.2 (Théorème de Cook) *Le problème* **SAT** *est* **NP***-complet.*

Une première restriction. On peut déjà se demander si une restriction des formules aux clauses à seulement 3 littéraux (le nouveau problème est appelé 3-**SAT**) nous permet de changer la complexité du problème. La réponse est négative.

Proposition 28.8 *3-***SAT** *est NP-complet.*

B Problème 2-SAT

On décide ici de restreindre un peu plus le problème.

Définition 28.11 (2-SAT) *Étant donné une formule de la forme $\varphi = \bigwedge_{i=1}^{n} l_i^1 \vee l_i^2$, où les l_i^j sont ds littéraux, φ est-elle satisfiable ?*

On a alors le résultat suivant :

Théorème 28.3 *Le problème 2-**SAT** est décidable en temps linéaire.*

Développement 14. Preuve du théorème ci-dessus.

C $\mathbf{MAX - SAT}$

On peut aussi définir un problème d'optimisation, le but étant cette fois de maximiser le nombre de clause valide.

Définition 28.12 ($\mathbf{MAX - SAT}$) *Étant donné une formule φ sous forme normale conjonctive avec les clauses $c_1, ..., c_m$, trouver*

$$\max_v |\{1 \leq i \leq m | v(c_i) = 1\}$$

Le problème précédent est évidemment NP-complet. Cependant, il existe des résultats intéressant sur le nombre de résultat et les algorithmes probabilistes.

Théorème 28.4 *Étant donné m clauses $c_1, ..., c_m$ ayant $k_1, ..., k_m$ littéraux respectivement. En notant $k = \min k_i$, il existe une valuation qui satisfait au moins*

$$\sum_{i=1}^{m} (1 - 2^{-k_i}) \geq m(1 - 2^{-k})$$

clauses.

La preuve du problème précédent nous donne aussi un algorithme d'approximation probabiliste pour le problème $\mathbf{MAX - SAT}$.

IV Compacité

On termine ici par un résultat de compacité, permettant de déduire des résultats sur un ensemble infini à partir de ses sous-ensemble finis. On suppose ici l'ensemble des variables propositionnelles dénombrable.

Définition 28.13 *Un ensemble de formule est satisfiable s'il existe une valuation qui satisfait toutes ses formules.*

Un ensemble de formule est dit **finiment** *satisfiable si toute ses parties finies sont satisfiables.*

Théorème 28.5 (Théorème de compacité de la logique propositionnelle) *Un ensemble de formules Σ est satisfiable si, et seulement si, Σ est finiment satisfiable.*

Développement 15. Preuve du théorème de compacité.

Application. On peut en déduire un résultat de compacité sur du coloriage de graphe.

Proposition 28.9 *Un graphe G est k-coloriable si, et seulement si, tous ses graphes finis le sont.*

Leçon 29

Langages rationnels et automates finis. Exemples et applications.

Auteur·e·s: Marin Malory
Niveau : MPI-L3
Pré-requis : Algorithmique
Références : [Sipser, 2013], [Carton and Perrin, 2008]

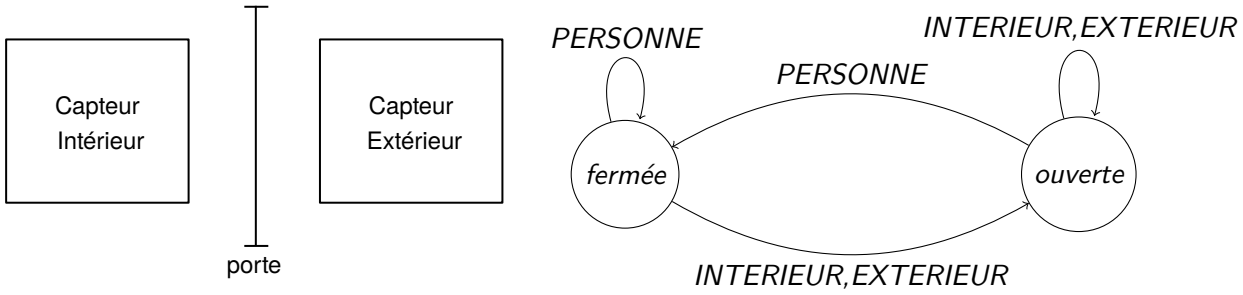
La théorie de la calculabilité commence par une question : qu'est-ce qu'un ordinateur ? Puisque les ordinateurs réels sont très compliqués, on utilise un **modèle de calcul** qui est simplement un ordinateur idéalisé bien défini mathématiquement. On commence ici avec le modèle le plus simple : **l'automate fini**.

I Automates finis

A Un premier exemple

L'idée d'un automate fini est de représenter un ordinateur ayant une quantité très limité de mémoire. On peut donner l'exemple d'un tel logiciel avec un simple mécanisme de gestion d'ouverture de porte automatique.

Exemple 29.1 (Porte automatique)



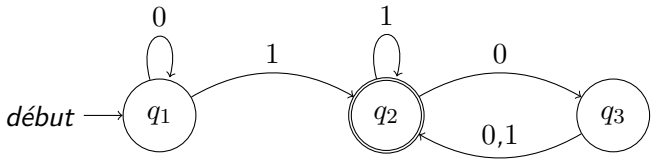
B Automate fini et calcul

Définition 29.1 *Un automate fini est un quintuplet* $(Q, \Sigma, I, F, \delta)$ *où*

- Q *est un ensemble fini d'***états** ;
- Σ *est un ensemble fini appelé* **alphabet** ;
- $\delta : Q \times \Sigma \to Q$ *est la* **fonction de transition** ;
- $q_0 \in Q$ *est l'***état initial** ;
- $F \subset Q$ *est l'ensemble des* **états acceptants**.

Exemple 29.2 *Un automate $M_1 = (Q, \Sigma, \delta, q_1, F)$ où $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{Q_2\}$, et δ décrit par le tableau ci-dessous.*

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2



Étant donné un automate fini $M = (Q, \Sigma, \sigma, q_0, F)$ et $w = w_1...w_n$ un mot sur Σ, on dit que M accepte w s'il existe une suite d'états $r_0, ..., r_n$ vérifiant :

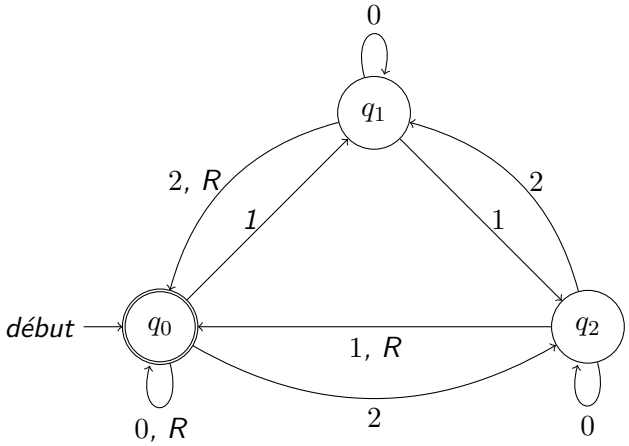
— $q_0 = r_0$;
— $\delta(r_i, w_{i+1}) = r_{i+1}$ pour $i = 0, ..., n-1$ et
— $r_n \in F$.

Étant donné un langage $L \subset \Sigma^*$, on dit que M reconnaît L si $L = \{w | M \text{ accepte } w\}$.

Définition 29.2 (Langage rationnel) *Un langage est dit* **rationnel** *s'il existe un automate fini M qui le reconnaît.*

Exemple 29.3 *On revient à notre exemple M_1. Ici, on a $L(M_1) = \{w \in \{0,1\}^* | w$ contient au moins un 1 et un nombre*

Exemple 29.4 (Un automate plus compliqué) *On construit un automate M_2 qui, sur l'alphabet $\Sigma = \{0, 1, 2, R\}$, qui étant donné un mot de Σ^* accepte si et seulement si la somme des éléments après le dernier R (reset) vaut 0 modulo 3.*



C Automate minimal

Le but est ici de construire un automate ayant le moins d'état possible.

Définition 29.3 *Soit $A = (Q, \Sigma, q_0, F, \delta)$ un automate déterministe complet. L'***équivalence de Nérode** *est une relation d'équivalence définie sur Q par*

$$q \sim_A q' \Leftrightarrow \{w \in \Sigma^* : \delta^*(q, w) \in F\} = \{w \in \Sigma^* : \delta^*(q', w) \in F\}$$

Autrement dit, q et q' ne sont pas équivalents si et seulement si on peut les distinguer par un mot w, c'est-à-dire si w est accepté depuis seulement un des deux.

Exercice 29.1

1. *Montrer que l'équivalence de Nérode est bien une relation d'équivalence.*

2. *L'équivalence de Nérode est régulière à droite, c'est-à-dire que si* $q \sim_A q'$ *et si* $a \in \Sigma$, *alors* $\delta(q,a) \sim_A \delta(q',a)$.

De l'exercice précédent on déduit que, en notant $\pi : Q \to Q/\sim_A$ la projection canonique, l'application $\overline{\delta} : (\pi(q),a) \to \pi(\delta(q,a))$ est bien définie.

Définition 29.4 (Automate minimal) *Soit* $A = (Q,\Sigma,q_0,F,\delta)$ *un automate déterministe complet. On appelle* **automate minimal** *de* A *l'automate* $\overline{A} = (Q/\sim_A,\Sigma,\pi(q_0),\pi(F),\overline{\delta})$.

Proposition 29.1 *Parmi tous les automates déterministes complets reconnaissant* $L(A)$*n l'automate* $\overline{A}$ *est l'unique (à renommage des états près) automate ayant le plus petit nombre d'états.*

Les classes d'équivalences de $\sim_A$ peuvent être déterminées effectivement en temps polynomial via **l'algorithme de Moore** ou de **Hopcroft**. on peut alors calculer en temps polynomial l'automate minimal d'un automate déterministe complet donné.

On peut aussi qualifier l'équivalence de Nérode via les **résiduels** mais cette fois sur les mots et non les états. Le résiduel du langage L par rapport au mot u est l'ensemble $u^{-1}L = \{v \in \Sigma^* | uv \in L\}$. On note alors $x \equiv_M y$ si, et seulement si $x^{-1}L = y^{-1}L$.

Théorème 29.1 (de Myhill-Nérode) *Un langage est rationnel si, et seulement si, il admet un nombre finis de résiduels.*

Développement 16. Preuve du théorème de Myhill-Nérode et application au lemme de non pompage.

D Application : recherche de motif

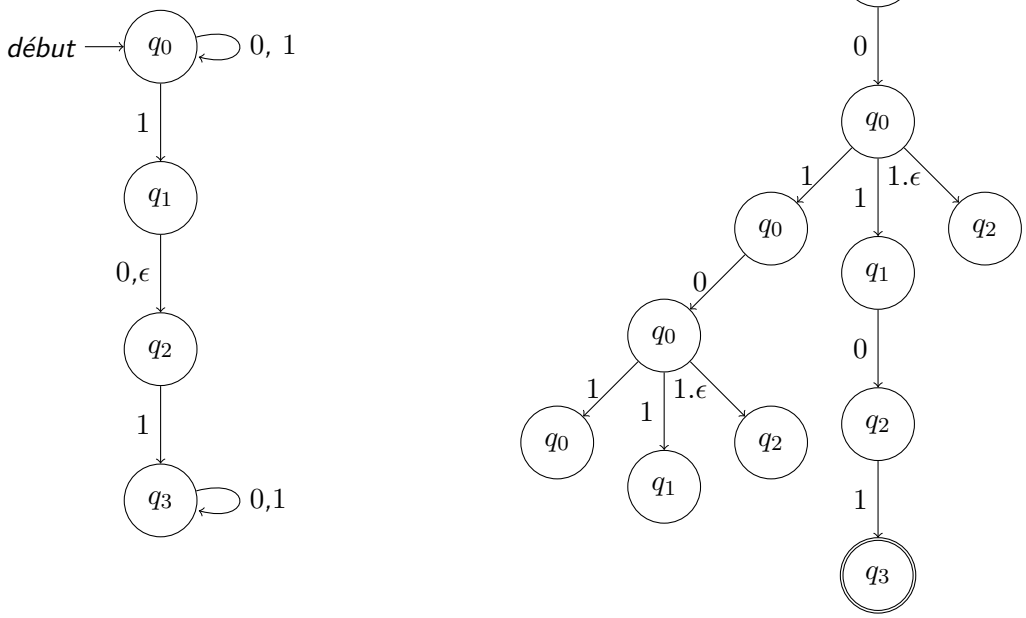
Les automates finis peuvent avoir des applications en algorithmique de texte. Étant donné un texte T de longueur n et un motif M de longueur m, l'algorithme naïf décidant si M est une sous-chaîne de T est un $\mathcal{O}(nm)$. On peut mieux faire avec des automates.

Développement 8. recherche de motif à l'aide de l'automate des occurrences.

II Non déterminisme

Jusqu'à présent, chaque étape de calcul dépend uniquement de l'étape précédente. Dans une machine non-déterministe, plusieurs choix peuvent exister.

Exemple 29.5 (automate non déterministe et d'un arbre de calcul) *On définit un automate non-déterministe* N_1 *qu'on simule sur l'entrée* 0101.



A Automates finis non déterministes

Définition 29.5 *Un automate fini non déterministe est un quintuplet $(Q, \Sigma, I, F, \delta)$ où*

- *Q est un ensemble fini d'***états** *;*
- *Σ est un ensemble fini appelé* **alphabet** *;*
- *$\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ est la* **fonction de transition** *où $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$;*
- *$q_0 \in Q$ est l'***état initial** *;*
- *$F \subset Q$ est l'ensemble des* **états acceptants**.

Exercice 29.2 *Donner un automate non-déterministe et déterministe qui reconnaît le langage sur l'alphabet unaire $\{1\}$ des mots ayant soit un nombre de 0 qui est un multiple de 2 ou 3.*

Théorème 29.2 (Équivalence) *Tout automate fini non déterministe est équivalent à un automate fini déterministe.*

Remarque 29.1 *La preuve du théorème précédent nous donne un algorithme de déterminisation d'un automate non déterministe.*

B Stabilité

Définition 29.6 (Opérations rationnelles) *Soient A et B deux langages. Les opérations rationnelles sont :*

- *l'***union** *: $A \cup B = \{w | w \in A \wedge w \in B\}$;*
- *la* **concaténation** *: $A.B = \{xy | x \in A \vee y \in B\}$;*
- *l'***étoile** *: $A^* = \bigcup_{k \geq 0} A^k$.*

Proposition 29.2 *La classe des langages rationnels est stable par opération rationnelle, par intersection et par passage au complémentaire.*

Remarque 29.2 *Les langages rationnels ne sont pas stables par inclusion ($a^n b^n \subset \Sigma^*$)*

III Expressions régulières

On va maintenant voir le penchant descriptif des langages rationnelles : les expressions rationnelles.

A Définition

Définition 29.7 (Expression régulière et langage associé)
L'ensemble des expressions régulières se définit inductivement par :

1. *$a \in \Sigma$, $\emptyset$ et ϵ sont des expressions régulières ;*
2. *si e_1 et e_2 sont des expressions régulières, alors $(e_1 \cup e_2)$, $(e_2.e_2)$ et (e_1^*) sont des expressions régulières.*

À chaque expression régulière, on associe un langage en prenant $\{a\}$, $\emptyset$ et $\{\epsilon\}$ dans le cas 1, et en suivant les opérations dans le cas 2.

Remarque 29.3 *On pourra confondre une expression rationnelle e et son langage associé $L(e)$.*

Exemple 29.6 *Étant donné l'alphabet $\Sigma = \{0, 1\}$:*

1. *$0^*10^* = \{w | w$ contient un unique $1\}$;*
2. *$\Sigma^*001\Sigma^* = \{w | w$ contient la sous-chaîne $001\}$*

Exemple 29.7 *L'ensemble des identifiants de la forme `id.N` où `N` est un nombre quelconque est le langage décrit par l'expression rationnelle `id.(1|2|...|9)*`. Ces expressions rationnelles sont très utiles pour décrire et trouver des chaînes vérifiant un pattern.*

Théorème 29.3 (Kleene 1956) *Un langage est rationnel si et seulement s'il est décrit par une expression rationnelle.*

Exemple 29.8 *La preuve de ce théorème nous fournit des algorithmes permettant de passer d'une expression rationnelle à un automate (méthode de Thompson) et vice-versa (algorithme de McNaughton et Yamada)*

Remarque 29.4 *Pour passer d'un automate à une expression rationnelle, on peut utiliser une méthode similaire à l'élimination de Gauss en mathématiques. Il faudrait pour cela un moyen de résoudre une équation entre langages, via le* **lemme d'Arden**.

B Application : analyse lexicale

Le lien entre expression rationnelle et automate finis permet notamment de créer un **analyseur lexical**.

IV Un modèle de calcul

A Des langages non rationnels ?

Proposition 29.3 (Lemme de l'étoile) *Pour tout langage rationnel L, il existe un entier n tel que pour tout $w \in L$ vérifiant $|w| \geq n$, w peut s'écrire $w = xyz$ avec :*

1. *pour tout $i \geq 0$, $xy^iz \in L$;*
2. *$|y| > 0$;*
3. *$|xy| \leq n$.*

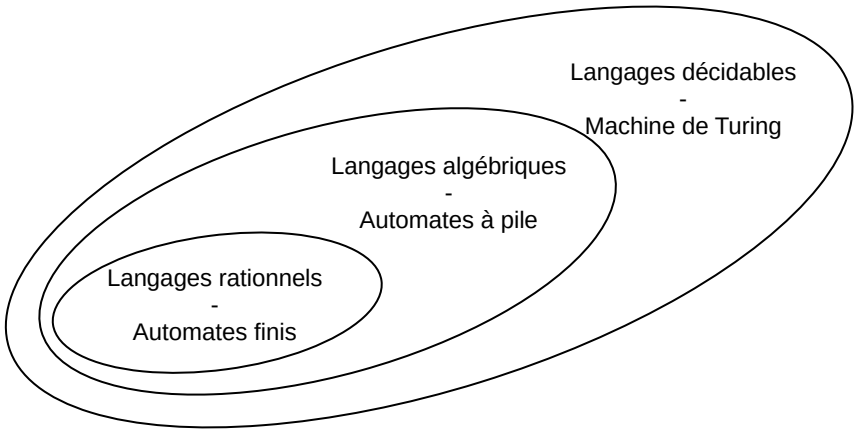
Exemple 29.9

— $L = \{a^n b^n, n \in \mathbb{N}\}$ *n'est pas rationnel car il ne vérifie pas le lemme précédent.*

— $L_{()}$ *l'ensemble des mots bien parenthésés non plus.*

B Hiérarchie de Chomsky

Les automates finis ne sont pas le seul modèle de calcul. Les automates finis et les langages rationnels forment ensemble le premier niveau de la **hiérarchie de Chomsky**.



La notion de décidabilité utilise les Machine de Turing, permettant de donner une définition formelle à la notion d'algorithme. On peut simplement définir la décidabilité d'un langage par l'existence d'un algorithme permettant de décider l'appartenance d'un mot à ce langage.

Théorème 29.4 (Presburger, 1929) *La théorie du premier ordre des entiers munis de l'addition est décidable.*

Deuxième partie

Développements

Développement 1

Correction du balayage de Graham

Auteur·e·s: Marin Malory
Références : [Cormen et al., 2009]

Ce développement consiste à montrer la correction d'un algorithme itératif, le balayage de Graham. Il trouve ainsi sa place tout naturellement dans la leçon 1 afin d'illustrer la notion d'invariant de boucle sur un algorithme complexe. Toute la subtilité de l'algorithme repose sur une gestion astucieuse d'une pile, ce qui permet de placer ce développement dans la leçon 5. Enfin, un pré-traitement est nécessaire afin de trier tous les points par angle polaire dont on veut calculer l'enveloppe convexe, permettant d'illustrer la leçon 8.

Balayage de Graham. On considère un ensemble de points Q, dont on veut calculer l'enveloppe convexe $\mathbf{EC}(Q)$.Le balayage de Graham résout le problème de l'enveloppe convexe en gérant une pile S de points candidats. Chaque point de l'ensemble Q est empilé une fois, et les sommets qui ne sont pas dans $\mathbf{EC}(Q)$ finissent par être dépilés.

On utilisera deux autres opérations sur la pile :

— `Sommet(S)` : retourne le sommet de la pile sans changer son contenu ;

— `Sous-Sommet(S)` : retourne l'élément juste en dessous du sommet de la pile, sans changer son contenu.

À la fin de l'exécution, S contiendra, du bas vers le haut, les sommets de $\mathbf{EC}(Q)$ dans l'ordre trigonométrique.

Algorithme. On suppose que l'on dispose d'une fonction `Tri-Polaire`(Q, p_0) qui trie les éléments de Q par angle polaire respectivement à p_0, dans le sens trigonométrique. Si deux éléments ont le même angle, on garde seulement celui qui est le plus loin de p_0.

Il sera judicieux d'illustrer l'algorithme sur un exemple simple.

Correction. On montre alors la correction partielle de notre algorithme en exhibant un invariant.

Théorème 1.1 *Si la procédure* `BalayageGraham` *est exécutée sur un ensemble Q de points tel que $|Q| \geq 3$, alors à la fin de la procédure, la pile S contient du bas vers le haut, les sommets de $\mathbf{EC}(Q)$ dans le sens trigonométrique.*

Démonstration. Pour $1 \leq i \leq m$, on pose $Q_i = \{p_1, ..., p_i\}$. Quelques remarques :

— $Q \backslash Q_m$ est l'ensemble des points supprimés par `Tri-Polaire`.

— $\mathbf{EC}(Q) = \mathbf{EC}(Q_m)$.

Algorithme 1.1 : BalayageGraham(Q)

```
p_0 ← sommet de S minimal pour l'ordre lexicographique sur (ordonnée, abscisse). ;
p_1, ..., p_m ← Tri-Polaire(Q\{p_0}, p_0) ;
S ← PileVide() ;
Empiler(S, p_0) ;
Empiler(S, p_1) ;
Empiler(S, p_2) ;
pour i = 3...m faire
    q_1 ← Sous-Sommet(S) ;
    q_2 ← Sommet(S) ;
    tant que (q_1, q_2, p_i) > 0 faire
        Dépiler(S) ;
        q_1 ← Sous-Sommet(S) ;
        q_2 ← Sommet(S) ;
    Empiler(S, p_i) ;
retourner S
```

— pour tout $1 \leq i \leq m$, $p_0, p_1, p_i \in \mathbf{EC}(Q_i)$

On montre alors l'invariant suivant :

« Au début de chaque itération du `Pour`, la pile S contient, du bas vers le haut, les sommets de $\mathbf{EC}(Q_{i-1})$ pris dans l'ordre trigonométrique. »

Initialisation Pour $i = 3$, on a $Q_{i-1} = \{p_0, p_1, p_2\}$. L'enveloppe convexe d'un triangle est lui-même et S contient p_0, p_1, p_2.

Conservation Au début de l'itération i, le sommet de la pile est p_{i-1}. Soit p_j le sommet de la pile après l'exécution du `Tant que` juste avant d'empiler p_i. En notant P_i l'ensemble des sommets dépilés lors de la boucle `Tant que` de l'itération i, on veut montrer que :

$$\mathbf{EC}(Q_j \cup \{p_i\}) = \mathbf{EC}(Q_i \backslash P_i) = \mathbf{EC}(Q_i) \tag{1.1}$$

Soit p_t un sommet dépilé, et soit p_r le sommet qui était juste en dessous de p_t. On sait deux choses :

— l'angle polaire de p_t (respectivement à p_0) est supérieur à celui de p_r et inférieur à p_i ;

— l'angle (p_r, p_t, p_i) est positif.

On en déduit alors que p_t est dans le triangle p_0, p_r, p_i et donc n'est pas dans $\mathbf{EC}(Q_i)$ (sauf s'il est sur le segment $[p_i; p_r]$, mais ce cas ne pose pas problème). Ainsi, on a :

$$\mathbf{EC}(Q_i \backslash \{p_t\}) = \mathbf{EC}(Q_i)$$

et en répétant ce raisonnement pour tous les points de P_i, on a

$$\mathbf{EC}(Q_i \backslash P_i) = \mathbf{EC}(Q_i)$$

Or, par définition, on a $Q_i \backslash P_i = Q_j \cup \{p_i\}$ et donc on en déduit l'égalité 1.1.

Cette égalité nous permet de justifier que S contient exactement les sommets de $\mathbf{EC}(Q_i)$, et l'ordre est trivial.

Terminaison Quand la boucle termine, on a $i = m + 1$ et donc S contient les sommets de $\mathbf{EC}(Q_m)$ dans l'ordre trigonométrique du bas vers le haut.

□

Complexité. Le balayage de Graham réalise $\mathcal{O}(n \log n)$ opérations élémentaires où n est le nombre de points. La recherche de p_0 est linéaire, et le tri se fait en $\mathcal{O}(n \log n)$. Le traitement suivant se fait alors en $\mathcal{O}(n)$ en complexité amortie. En effet, chaque sommet est empilé exactement une fois et chaque sommet ne peut être dépilé qu'une fois. Au total, on réalise exactement m fois `Empiler` et au plus $m-2$ fois `Dépiler` (car p_0, p_1 et p_m ne sont jamais dépilés). Puisque $m \leq n$, on réalise tout le traitement en $\mathcal{O}(n)$ opérations.

Développement 2

Optimalité du glouton sur les matroïdes

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Ce développement consiste à donner une brève introduction à la théorie des matroïdes. On montre que l'algorithme glouton est optimal pour ces structures, s'intégrant ainsi dans la leçon 1 et 12. La principale application de cette théorie étant la preuve de l'optimalité de l'algorithme de Kruskal, ce développement peut aussi illustrer la leçon 10.

Définition 2.1 (Matroïde, Whitney, 1935)
Le couple (S, I) est un matroïde si S est un ensemble à n éléments, $I \subset P(S)$ et si :

1. $X \in I \Rightarrow (\forall Y \subset X, X \in I)$ *(hérédité) ;*
2. $(A, B \in I, |A| < |B|) \Rightarrow \exists x \in B \backslash A, A \cup \{x\} \in I$ *(propriété d'échange).*

Tout ensemble $X \in I$ est dit **indépendant**.

Exemple de matroïde. On peut montrer que l'ensemble des forêts d'un graphe est un matroïde.

Théorème 2.1 *Soit $G = (V, E)$ un graphe, $S = E$ et $I = \{A \subset E | A$ ne contient pas de cycle$\}$. Le couple (S, I) est un matroïde.*

Démonstration. Un ensemble $X \in I$ est indépendant si, et seulement si, X est une forêt du graphe G.

1. Un sous-ensemble d'une forêt est encore une forêt. En effet, si $X \subset I$ et il existe $Y \subset X$ contenant un cycle, alors X contient aussi ce cycle, ce qui est absurde.
2. Soient A et B deux forêts de G avec $|A| < |B|$. Chaque sommet de G est contenu dans un arbre de A (si le sommet est isolé, alors il est dans un arbre à un sommet). Ainsi, A contient $|V| - |A|$ arbres (resp. $|V| - |B|$ pour B). En effet, dès qu'on ajoute une arête, on fusionne deux arbres et le nombre total d'arbres diminue de un.

   Ainsi, B contient moins d'arbres que A, et donc il existe un arbre T de B qui n'est pas inclus dans un arbre de A (si ce n'était pas le cas, on aurait $|V| - |B| \geq |V| - |A|$ car chaque arbre de B est dans un arbre de A). Autrement dit, il existe deux sommets $u, v \in V$ tels que u et v sont dans T mais pas dans le même arbre de A. Il existe donc une arête (x, y) sur le chemin de T allant de u à v qui n'est pas dans A. On a alors $A \cup \{(x, y)\}$ qui est toujours une forêt.

□

Ensemble indépendant maximal. On introduit ici la notion d'ensemble indépendant maximal et de matroïde pondéré.

Définition 2.2 *Soit $F \in I$. $x \notin F$ est une* **extension** *de F si $F \cup \{x\} \in I$. Un ensemble indépendant est* **maximal** *s'il n'a aucune extension.*

Lemme 2.1 *Tous les ensembles indépendants maximaux d'un matroïde ont le même cardinal.*

Démonstration. Par l'absurde, on utilise la propriété d'échange et on contredit la maximalité du plus petit des deux ensembles indépendants. □

Définition 2.3 (Matroïde pondéré) *Un matroïde pondéré est un matroïde (S, I) muni d'une fonction de poids $w : S \to \mathbb{N}$. Le poids d'un ensemble $X \subset S$ est défini par $w(X) = \sum_{x \in X} w(x)$.*

Algorithme glouton sur un matroïde pondéré. On cherche à trouver un ensemble indépendant de poids maximum. On peut montrer que dans le cas d'un matroïde, l'algorithme glouton est optimal.

Algorithme 2.1 : GloutonMatroide(S,I, w)

Données : matroïde (S, I) avec $S = \{s_1, ..., s_n\}$ trié par poids décroissants
Résultat : Ensemble indépendant A de poids maximum.
$A \leftarrow \emptyset$;
pour $i = 1...n$ **faire**
  si $A \cup \{s_i\} \in I$ **alors**
    $A \leftarrow A \cup \{s_i\}$;
retourner A

Théorème 2.2 *L'algorithme* `GloutonMatroïde` *renvoie une solution optimale.*

Démonstration.

Soit $X = \{x_1, ..., x_m\}$ la solution renvoyée par l'algorithme glouton. On a alors $w(x_1) \geq ... \geq w(x_m)$. Soit $Y = \{y_1, ..., y_m\}$ une solution optimale avec $w(y_1) \geq ... \geq w(y_m)$. On montre que pour tout $1 \leq i \leq m$, $w(x_i) \geq w(y_i)$. L'optimalité de la solution en découle directement.

Si ce n'était pas le cas, prenons k le plus petit indice tel que $w(x_k) < w(y_k)$. On remarque que $k > 1$ car $\{x_1\}$ est le singleton indépendant de poids maximum. On regarde alors $A = \{x_1, ..., x_{k-1}\}$ et $B = \{y_1, ..., y_k\}$. Puisque $|B| = |A| + 1$, on peut appliquer la propriété d'échange et il existe $1 \leq i \leq k$ tel que $A \cup \{y_i\}$ est indépendant et $y_i \notin A$. On a alors $w(y_i) \geq w(y_k) > w(x_k)$. L'algorithme glouton aurait alors choisit y_i avant x_k (il suffit de prendre le sous-ensemble de $A \cup \{y_i\}$ des éléments ayant un poids inférieur à $w(y_j)$, qui est indépendant par hérédité). □

Optimalité de l'algorithme de Kruskal. Le théorème précédent nous permet de montrer que l'algorithme de Kruskal qui construit un arbre couvrant renvoie bien un arbre optimal. En effet, les arêtes sont triées par poids croissant et on choisit de manière gloutonne la prochaine arête qui ne crée pas de cycle.

Développement 3

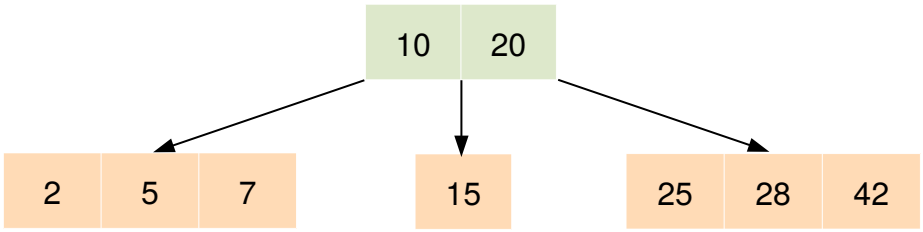
B-arbres

Auteur·e·s: Marin Malory
Références : [Cormen et al., 2009]

Ce développement a pour but de présenter une structure de donnée arborescente implémentant un dictionnaire tout en prenant en compte la hiérarchie mémoire. Il trouve alors tout naturellement sa place dans les leçons 4, 6, 10 et 15. Si la leçon 16 parle de hiérarchie mémoire, ce développement peut s'y intégrer. Enfin, avec un point de vue base de données, les B-arbres peuvent servir en indexation afin d'optimiser certaines requêtes. En modifiant la mise en contexte, ce développement s'intègre alors dans les leçons 18, 22 et 23.

Motivation et exemple. Dans le cas d'une grande quantité d'information, il est parfois nécessaire d'utiliser des supports de stockage dont le temps de réponse pour une lecture est élevé (disque dur, ...). Pour implanter une structure de dictionnaire dans ce cas, on préférera contracter notre ABR afin de faire moins d'appels à la mémoire lente.

Exemple 3.1 (Un B-arbre d'ordre 2 et de hauteur 2)



Définition.

Définition 3.1 (B-arbre) *Soit un entier $t \geq 2$. On appelle B-arbre d'ordre t un arbre vérifiant les invariants suivants :*

— *Chaque nœud x contient les attribut suivants :*

1. *le booléen x.feuille (VRAI si X est une feuille, FAUX sinon) ;*
2. *le nombre $x.n$ de clés dans ce nœud ;*
3. *le tableau trié x.clé de taille $x.n$ contenant les clés de ce noeud (numéroté de 1 à $x.n$) ;*
4. *le tableau x.fils de taille $x.n + 1$ contenant l'ensemble des enfants du nœud x (numéroté de 0 à $x.n$).*

— *pour tout nœud x, toute clé k_i apparaissant dans le fils $x.\text{fils}_i$, on a $k_{i-1} \leq x.\text{clé}_i \leq k_i$ $(1 \leq i \leq n)$.*

— *toutes les feuilles ont la même profondeur h ;*

— *pour tout noeud x non racine, on a $t-1 \leq x.n \leq 2t-1$;*

— *la racine x_0 vérifie $1 \leq x_0.n \leq 2t-1$.*

Ici, au vu de son utilisation, on suppose le B-arbre stocké dans une mémoire auxiliaire d'accès lent, on va donc compter notre complexité en fonction du nombre de lecture et d'écriture dans cette mémoire. On considère deux fonctions $\mathbf{LIRE}$ et $\mathbf{ECRIRE}$ permettant d'accéder à notre mémoire auxiliaire.

Recherche dans un tel arbre. La fonction de recherche dans un B-arbres ressemble beaucoup à la recherche dans un ABR, sauf que dans chaque nœud, il faut chercher le bon intervalle, et que la complexité sera évaluée en fonction d'appel à la fonction $\mathbf{LIRE}$. On suppose que notre racine est en mémoire principale, il n'y a donc pas besoin de faire de appel à $\mathbf{LIRE}$.

Algorithme 3.1 : Recherche(x,c)

Données : un noeud x et une clé c
Résultat : si c est dans un nœud y de l'arbre, on renvoie y et sa place dans le nœud, sinon on renvoie NIL.

```
# Recherche du bon intervalle ;
trouve, i ← Dichotomie(x.clé, c) ;
si trouve alors
    retourner x, i
si x.feuille alors
    retourner NIL
f ← LIRE(x.fils_{i-1}) ;
retourner Recherche(f, c)
```

On va maintenant étudier la complexité de notre algorithme et comparer à la complexité si on avait utiliser un ABR, c'est-à-dire un B-arbre d'ordre 2.

Théorème 3.1 *Dans un B-arbre T d'ordre $t \geq 2$ contenant n clés, sa hauteur h vérifie*

$$h \leq \ln_t\left(\frac{n+1}{2}\right)$$

Démonstration. Par définition, T contient au moins une clé à la racine, et donc sa racine possède au moins 2 enfants. On a donc :

— au moins 1 nœud à la profondeur 0 ;

— au moins 2 nœuds à la profondeur 1 ;

— au moins $2t$ nœuds à la profondeur 2...

On montre alors par récurrence immédiate qu'on a au moins $2t^{i-1}$ nœuds à la profondeur i. Puisque chaque nœud contient au moins $(t-1)$ clés, on a :

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h-1}{t-1}\right) = 2t^h - 1$$

Le résultat en découle alors directement. □

Étant donné un B-arbre T de racine x_0, la fonction Recherche(x,c) fait h appels à $\mathbf{LIRE}$ dans le pire des cas, que l'on peut borner via le théorème ci-dessus.

Insertion dans un B-arbre. Ici les choses se compliquent puisque l'insertion est moins évidente. Il y a des situations qui peuvent poser problème, par exemple si l'arbre est déjà plein.

Idée de l'algorithme : sur l'entrée c et le nœud x,

— On va à la feuille correspondante en choisissant les chemins via les intervalles ;

— Si la feuille possède strictement moins de $2t-1$ étiquettes, on peut insérer c ;

— Sinon, en notant $c_1, ..., c_{2t}$ les nouvelles étiquettes de cette feuilles (avec $c = c_{2t}$), on sépare cette feuilles en deux feuilles d'étiquettes $c_1, ..., c_{t-1}$ et $c_{t+1}, ..., c_{2t}$, et on remonte c_t comme séparateur dans le parent ;

— Si le parent contient alors trop d'étiquettes, on le coupe en deux de la même manière et ainsi de suite jusqu'à la racine.

— Si la racine doit être coupée en deux, on crée une nouvelle racine qui ne contiendra qu'une étiquette.

Développement 4

Construction d'un tas en temps linéaire et tri par tas

Auteur·e·s: Marin Malory
Références : [Cormen et al., 2009]

Ce développement présente un algorithme permettant de construire un tas en temps linéaire, ainsi que le tri par tas. Un tas permet notamment d'implémenter une file de priorité, illustrant ainsi les leçons 4 et 5. De plus, ce développement illustre l'utilisation des arbres pour réaliser des structures de données arborescentes et s'insère donc naturellement dans la leçon 10.

Tas-max. La structure de tas max est une structure de données qu'on peut représenter par un arbre binaire complet gauche, où chaque étiquette d'un nœud est plus grande que celle de ses descendants. Un tas peut être implanter par un tableau à n éléments numérotés de 1 à n, où le fils gauche (resp. droit) d'un élément $i \in \{0, ..., n-1\}$ est $2i$ (resp. $2i+1$).

Tri par tas. L'idée du tri par tas est assez élémentaire une fois qu'on a la structure de donnée : étant donné les n éléments à trier, on construit un tas les contenant puis on extrait n fois le minimum du tas.

Étant donné un tableau A (représentant un arbre binaire) et un indice i, on note T_i l'arbre correspondant enraciné au noeud i. L'algorithme suivant, `Entasser-Max`, reçoit un tableau A et un indice i tel que les arbres enracinés en $2i$ et $2i+1$ soient des tas, et modifie A afin que T_i soit un tas.

Algorithme 4.1 : Entasser-Max(A,i)

$g \leftarrow 2i$;
$d \leftarrow 2i+1$;
$\max \leftarrow \text{argmin}_{j \in \{i,g,d\}} A[j]$;
si $\max \neq i$ **alors**
  $A[i] \leftrightarrow A[\max]$;
  Entasser-Max(A, $\max$) ;

Illustrer le fonctionnement de la procédure `Entasser-Max` *sur un exemple.*

On déduit de cet algorithme une procédure permettant de transformer un tableau en tas-max en temps linéaire. Enfin, avec ce tas, on en déduit une procédure permettant de réaliser un tri en place.

Illustrer le fonctionnement de la procédure `Tri-Par-Tas` *sur un exemple.*

Algorithme 4.2 : Construire(A)

pour $i = \lfloor n/2 \rfloor, ..., 1$ **faire**
  Entasser-Max(A,i) ;

Algorithme 4.3 : Tri-Par-Tas(A)

Construire(A) ;
pour $i = A.taille - 1, ..., 1$ **faire**
  $A[1] \leftrightarrow A[i]$;
  A.taille $\leftarrow A$.taille $- 1$;
  Entasser-Max(A,1) ;

Analyse. Le seul point complexe est la correction de la procédure `Entasser-Max`. Les autres corrections en découle presque directement.

Lemme 4.1 *Soit $1 \leq i \leq n$, si T_{2i} et T_{2i+1} sont des tas , alors après* `Entasser-Max(`A`,`i`)`, *T_i est un tas. La procédure s'exécute en temps $\mathcal{O}(h(i))$, où $h(i)$ est la hauteur de T_i.*

Démonstration. On note C_h la complexité dans le pire des cas de l'algorithme Tasser pour un nœud i dont l'arbre enraciné en i est de hauteur h. On a alors dans le pire cas $C_h = \mathcal{O}(1) + C_{h-1}$ et donc $C_h = \mathcal{O}(h)$.

Prouvons maintenant la correction. On montre par induction structurelle sur les arbres binaires que l'arbre obtenu vérifie la propriété du tas-max.

Si T_i est une feuille, alors `Entasser-Max(`A`,`i`)` ne modifie pas T_i qui est déjà un tas. Sinon, on fait une disjonction de cas selon la valeur de `plus_grand` :

— si `max`$=i$, alors l'arbre enraciné en i est un tas ;

— si `max`$=g$, alors on a si $d < n$ $A[g] \geq A[i]$ et $A[g] \geq A[d]$. On échange alors dans A les nœuds i et g et la propriété du tas-max est vérifiée localement. Puisque T_g était bien un tas par hypothèse, ses enfants le sont aussi et donc ont peut appliquer l'hypothèse d'induction sur g et après `Entasser-Max(`A`,`g`)`, T_g est aussi un tas.

— si `max`$=d$, on raisonne de la même manière.

□

Lemme 4.2 *La procédure* `Construire(`A`)` *transforme A en un tas en temps $\mathcal{O}(n)$.*

Démonstration.

Complexité : On note $C(n)$ la complexité dans le pire des cas de l'appel `Construire(`A`)` où $n = A$.taille. Uun tas à n éléments a une taille bornée par $\lfloor \log_2 n \rfloor$ et le nombre de nœud ayant la hauteur h est au plus $\lceil n/2^{h+1} \rceil$. Puisque notre fonction appelle la fonction `Tasser` pour tout nœud ayant une hauteur strictement positive, on a :

$$C(n) = \sum_{h=1}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(\sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} h \right)$$

Or, on peut calculer la deuxième somme en dérivant la série géométrique de raison x et en appliquant la formule en $1/2$. On obtient alors

$$\sum_{h \geq 0} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

et ainsi $C(n) = \mathcal{O}(n)$.

Correction : Le fait que l'arbre obtenue soit complet gauche est direct via la représentation par tableau. Il reste à montrer qu'il vérifie la propriété du tas-max. On utilise l'invariant de boucle suivant : « Au début de chaque itération de la boucle, T_{i+1}, T_{i+2}, ... , T_n sont des tas ».

— **Initialisation :** Avant la première itération, $i = \lfloor n/2 \rfloor$ et pour $j > \lfloor n/2 \rfloor$, alors T_j contient une unique feuille, et vérifie donc la propriété du tas-max trivialement.

— **Hérédité :** Pour voir que chaque itération conserve l'invariant, on observe que les enfants du nœud i ont des indices supérieurs à i. D'après l'invariant, ce sont donc tous les deux des racines d'un arbre vérifiant la propriété du tas-max. D'après le lemme 4.1, après `Entasser-Max(`A`,`i`)`, T_i. Ainsi, tous les arbres enracinés en i, $i+1$,etc sont des tas.

À la fin, on a $i = 0$ et donc l'arbre enraciné en 1 est un tas. Ainsi, A est un tas.

□

Théorème 4.1 *La procédure `TriParTas(`A`)` trie en place un tableau T à n éléments en temps $\mathcal{O}(n \log n)$.*

Démonstration.

Complexité : D'après le lemme 4.2, l'appel `Construire(`T`,`n`)` transforme T en un tas en temps $\mathcal{O}(\log n)$ sans prendre de place supplémentaire. De plus, l'algorithme fait $n-1$ appels échanges et appels à la fonction `Tasser` qui s'exécute dans le pire des cas en $\mathcal{O}(\log n)$. Donc l'algorithme s'exécute en $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$.

Correction : On pose l'invariant suivant : « à chaque itération, le sous-tableau $A[i+1...n]$ contient les $n-i-1$ plus grands éléments de T triées, et $A[1...i]$ est un tas max ».

— Avant de rentrer dans la boucle, on a $i = n$ et donc $A[n+1, n]$ est vide et $A[1...n]$ est effectivement un tas d'après le lemme 4.2.

— On suppose que pour $i > 0$, on a $A[i+1...n]$ trié et qui contient les plus grands éléments et $A[1...i]$ est un tas. On a alors $A[1]$ le plus grand élément de $A[1...i]$, et donc après échange, $A[i...n]$ est encore trié et contient les $n-i$ plus grands éléments de A. Ensuite, d'après le lemme 4.1, après l'appel `Entasser-Max(`A`,0)`, on a $A[1...i-1]$ qui est un tas.

À la fin de la boucle, on a alors $i = 0$ et donc $A[1...n]$ qui est trié. □

Développement 5

Complexité moyenne de la recherche dans une table de hachage

Auteur·e·s: Rousseau Guillaume, Marin Malory
Références : [Cormen et al., 2009]

Ce développement étudie la complexité moyenne de la recherche dans une table de hachage où les collisions sont gérées par chaînage. Il trouve ainsi sa place dans la leçon 6, et peut servir d'illustration dans la leçon 4. Ce développement étant un peu cours, il pourra être complété par une simple analyse du pire des cas au début, ainsi qu'avec des dessins au tableau afin d'illustrer les démonstrations.

Introduction. On considère une table de hachage où les collisions sont résolues par chaînage, et on se place dans l'hypothèse de **hachage uniforme simple** : chaque élément a la même chance d'être haché vers l'une des alvéoles indépendamment des endroits où les autres éléments sont allés.

Pour une table de hachage à m alvéoles et n éléments, on note $\alpha = n/m$ le facteur de remplissage.

Théorème 5.1 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche infructueuse prend un temps moyen $\Theta(1+\alpha)$, sous l'hypothèse d'un hachage simple uniforme.*

Démonstration. On note $\mathbf{Rech}^-(T,k)$ la variable aléatoire qui compte le nombre de comparaisons dans une recherche infructueuse d'un élément k dans un tableau T à n éléments.

$$\mathbb{E}(\mathbf{Rech}^-(T,k)) = \sum_{i=1}^{m} \mathbb{P}(h(k)=i)\lambda_i$$

où λ_i est la taille de l'alvéole i de T.

Par hypothèse de hachage uniforme simple,

$$\mathbb{E}(\mathbf{Rech}^-(T,k)) = \frac{1}{m}\sum_{i=1}^{m} \lambda_i = \alpha$$

Maintenant, on note $\mathbf{Rech}^-(n,m)$ le nombre de comparaison dans un tableau déjà rempli avec n éléments $k_1,...,k_n$. Par hypothèse de hachage uniforme simple, le tableau est tiré uniformément parmi l'ensemble des tableaux à m alvéoles et n éléments. Ainsi, par linéarité de l'espérance,

$$\mathbb{E}(\mathbf{Rech}^-(n,m)) = \frac{1}{m^n}\sum_{T} \mathbb{E}(\mathbf{Rech}^-(T,k))$$

et donc $\mathbb{E}(\mathbf{Rech}^-(n,m)) = \alpha$. □

Théorème 5.2 *Dans une table de hachage pour laquelle les collisions sont résolues par chaînage, une recherche fructueuse prend un temps moyen $\Theta(1+\alpha)$, sous l'hypothèse d'un hachage simple uniforme.*

Démonstration. L'idée est de prendre aléatoirement un élément qui est déjà dans le tableau, et compter le nombre de comparaisons nécessaires. Le nombre d'éléments examinés pour un élément x est 1 plus le nombre d'éléments suivants qui ont été hachés au même endroit.

On suppose que l'on a inséré les éléments $x_1, ..., x_n$ dans la table de hachage, de clés respectives $k_1, ..., k_n$. On note $\mathbf{Rech}^+(x_i)$ le nombre de comparaisons nécessaires pour trouver x_i. De plus, on note $X_{ij} = \mathbf{1}\{h(k_i) = h(k_j)\}$. Par hypothèse de hachage simple uniforme, $\mathbb{E}(X_{ij}) = 1/m$. Ainsi,

$$\begin{aligned}
\mathbb{E}(\mathbf{Rech}^+(n,m)) &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}(\mathbf{Rech}^+(x_i)) \\
&= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}\left(1+\sum_{j=i+1}^{n}X_{ij}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathbb{E}(X_{ij})\right) \\
&= 1+\frac{1}{n}\sum_{1\leq i<j\leq n}\frac{1}{m} \\
&= 1+\frac{1}{n}\frac{n(n-1)}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{1}{2m}
\end{aligned}$$

□

Développement 6

Analyse du tri rapide randomisé

Auteur·e·s: Marin Malory
Références : [Motwani and Raghavan, 1995]

Ce développement a pour but d'étudier la complexité moyenne du tri rapide randomisé, et s'intègre ainsi naturellement dans la leçon 8. Puisque cet algorithme est probabiliste et utilise le paradigme « diviser pour régner » , il peut aussi servir d'illustration dans les leçons 11 et12.

Le but est de montrer le théorème suivant, qui analyse le temps d'exécution moyen du tri rapide randomisé (pour le pseudo-code, voir 8.6).

Théorème 6.1 *L'espérance du nombre de comparaisons du tri rapide randomisé d'un ensemble à n éléments est au plus $2nH_n$ où H_n est le terme général de la série harmonique.*

Démonstration. Soit T un tableau à n éléments. Pour $1 \leq i \leq n$, on note t_i le i-ème plus petit élément de T. Ainsi, $t_1 = \min T$ et $t_n = \max T$. On pose X_{ij} la variable aléatoire valant 1 si t_i et t_j sont comparés au cours de l'exécution, et 0 sinon.

On remarque alors que deux éléments t_i et t_j sont comparés au plus une fois. En effet, si t_i et t_j sont comparés , l'un des deux était un pivot et n'est comparé avec plus personne ensuite.

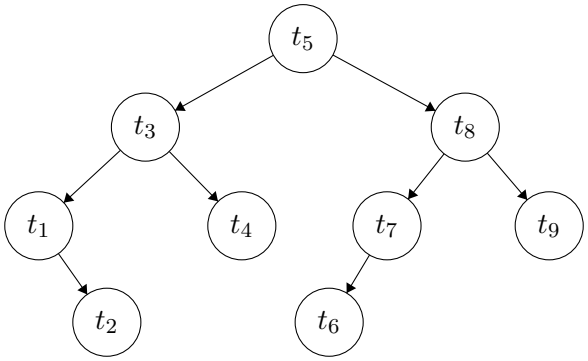
Ainsi, le nombre total de comparaisons est $N = \sum_{1 \leq i < j \leq n} X_{ij}$. Par linéarité de l'espérance, on a alors

$$\mathbb{E}(N) = \sum_{1 \leq i < j \leq n} \mathbb{E}(X_{ij})$$

En notant $p_{ij} = \mathbb{P}(X_{ij} = 1)$, on a directement $\mathbb{E}(X_{ij}) = p_{ij}$.

Lemme 6.1 *Pour tout $1 \leq i < j \leq n$, on a $p_{ij} = \frac{2}{j-i+1}$.*

On peut voir l'exécution du tri rapide comme un arbre binaire, avec comme nœuds les pivots successifs. On note alors π la permutation de T obtenu en faisant un parcours en largeur de l'arbre obtenu. L'arbre ci-dessous représente l'exécution du tri rapide sur un tableau à 9 éléments. On a ici $\pi = (4, 8, 2, 5, 1, 9, 6, 3, 7)$.



Démonstration. On peut faire deux remarques :

1. t_i et t_j sont comparés ssi pour tout $i < l < j$, $\pi(t_i) \leq \pi(t_l)$ et $\pi(t_j) \leq \pi(t_l)$. En effet, t_i et t_j ne sont pas comparés si, et seulement si, un pivot entre les deux les a séparés précédemment.
2. Chaque élément $t_i, ..., t_j$ a la même probabilité d'être le premier d'entre eux choisi pour être un pivot, et donc d'apparaître avant dans π.

Ainsi, en combinant les deux remarques, p_{ij} est exactement la probabilité que t_i ou t_j soit choisi en premier parmi $t_i, ..., t_j$ (1), et cette probabilité vaut $\frac{2}{j-i+1}$ (2). □

On peut alors revenir à notre formule,

$$\begin{aligned}
\mathbb{E}(N) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2nH_n
\end{aligned}$$

Or, on sait que $H_n \sim \ln n + o(1)$, le temps d'exécution de l'algorithme est $\mathcal{O}(n \log n)$. □

Développement 7

Distance d'édition

Auteur·e·s: Marin Malory
Références : [Crochemore and Rytter, 1994]

Ce développement présente la distance d'édition entre deux chaînes de caractères, ainsi qu'un algorithme de programmation dynamique permettant de la calculer, s'intégrant ainsi dans les leçons 9 et 12.

Définitions. Il y a plusieurs définition de distance possible entre deux chaînes de caractères. On va définir la distance d'édition, ou distance de Levenshtein, entre deux chaînes de caractères x et y en considérant les opérations suivantes :

- la **substitution** d'une lettre de x à une position donnée par une lettre de y ;
- la **suppression** d'une lettre de x à une position donnée ;
- l'**insertion** d'une lettre de y dans x à une position donnée.

Pour chaque opération, on définit un coût. Soit $a, b \in \Sigma$, on a :

- $\mathbf{sub}(a, b)$: coût pour substituer b par a ;
- $\mathbf{sup}(a)$: coût pour supprimer la lettre a ;
- $\mathbf{ins}(b)$: coût pour insérer la lettre b.

Définition 7.1 *Étant donné deux chaînes de caractères x et y, la distance d'édition entre x et y, noté $\mathbf{lev}(x, y)$ est défini par :*

$$\mathbf{lev}(x, y) = \min\{\text{coût de } \sigma : \sigma \in T_{x,y}\}$$

où $T_{x,y}$ est l'ensemble des séquences d'opérations qui transforme x en y, le coût d'une séquence étant la somme des coût de chaque opération.

Remarque 7.1

1. *La distance de Hamming est un cas particulier de la distance d'édition, où on considère seulement l'opération de substitution (on peut mettre un coût valant $+\infty$ pour ces deux opérations).*
2. *Ici, on se contentera de prendre les coût $\mathbf{sup}$ et $\mathbf{ins}$ unitaire, et $\mathbf{sub}(a, b) = \mathbf{1}\{a \neq b\}$.*

Exemple 7.1 (ADN) *On considère deux mots sur l'alphabet $\{A, T, C, G\}$, $x = AATGC$ et $y = CAGC$. On a $\mathbf{lev}(x, y) = 2$ puisque qu'il suffit de substituer le premier A de x à C, et supprimer le T.*

Calcul de la distance. On va donner un algorithme de programmation dynamique pour calculer la distance d'édition. Étant donné $x, y \in \Sigma^*$ de taille m et n respectivement, on définit le tableau T à $m+1$ lignes et $n+1$ colonnes tel que :

$$T[i, j] = \mathbf{lev}(x[0...i], y[0...j])$$

pour $i \in \{-1, 0, ..., m-1\}$ et $j \in \{-1, 0, ..., n-1\}$. Pour calculer $T[i,j]$ on utilisera la proposition suivante :

Proposition 7.1 *Pour $i = 0, ..., m-1$ et $j = 0, ..., n-1$, on a*

$$\begin{array}{lcl} T[-1,-1] & = & 0 \\ T[i,-1] & = & T[i-1,-1] + \mathbf{supp}(x[i]) \\ T[-1,j] & = & T[-1,j-1] + \mathbf{ins}(y[j]) \\ T[i,j] & = & \min \left\{ \begin{array}{l} T[i-1,j-1] + \mathbf{sub}(x[i],y[j]) \\ T[i-1,j] + \mathbf{sup}(x[i]) \\ T[i,j-1] + \mathbf{ins}(y[j]) \end{array} \right. \end{array}$$

On peut alors directement écrire l'algorithme et en déduire sa complexité.

Algorithme 7.1 : DynamicLev(x,y)

$T[-1,-1] \leftarrow 0$;
pour $i = 0...m-1$ **faire**
  $T[i,-1] \leftarrow T[i-1,-1] + \mathbf{sup}(x[i])$
pour $j = 0...n-1$ **faire**
  $T[-1,j] \leftarrow T[-1,j-1] + \mathbf{ins}(y[j])$;
  pour $i = 0...m-1$ **faire**
    $T[i,j] \leftarrow \min \left\{ \begin{array}{l} T[i-1,j-1] + \mathbf{sub}(x[i],y[j]) \\ T[i-1,j] + \mathbf{sup}(x[i]) \\ T[i,j-1] + \mathbf{ins}(y[j]) \end{array} \right.$;
retourner $T[m-1,n-1]$

Exemple 7.2 *On revient à notre exemple précédent en prenant chaque coût égal à 1 :*

	ϵ	A	A	T	G	C
ϵ	2	1	2	3	4	5
C	1	2	3	4	5	5
A	2	1	2	2	3	4
G	3	2	2	3	2	3
C	4	3	3	3	3	2

Correction de l'algorithme. On utilisera ce lemme qu'on utilisera pour montrer la proposition précédente.

Lemme 7.1 *Étant donné $a, b \in \Sigma$, $u, v \in \Sigma^*$, on a :*

$$\begin{array}{lcl} \mathbf{lev}(ua,\epsilon) & = & \mathbf{lev}(u,\epsilon) + \mathbf{sup}(a) \\ \mathbf{lev}(\epsilon,vb) & = & \mathbf{lev}(\epsilon,v) + \mathbf{ins}(b) \\ T[i,j] & = & \min \left\{ \begin{array}{l} \mathbf{lev}(u,v) + \mathbf{sub}(a,b) \\ \mathbf{lev}(u,vb) + \mathbf{sup}(a) \\ \mathbf{lev}(ua,v) + \mathbf{ins}(b) \end{array} \right. \end{array}$$

Démonstration. La séquence d'opérations qui transforme ua en ϵ peut être réarranger de telle sorte à finir par la suppression de la lettre a (commutativité de la suppression). Le reste de la séquence transforme u

en ϵ. Ainsi, on a :

$$\begin{aligned}\mathbf{lev}(ua,\epsilon) &= \min\{\text{coût de } \sigma : \sigma \in T_{ua,\epsilon}\}\\ &= \min\{\text{coût de } \sigma'.(a,\epsilon) : \sigma' \in T_{u,\epsilon}\}\\ &= \min\{\text{coût de}\sigma' : \sigma' \in T_{u,\epsilon}\} + \mathbf{sup}(a)\\ &= \mathbf{lev}(u,\epsilon) + \mathbf{sup}(a)\end{aligned}$$

Le second point se fait de la même manière. Pour le troisième point, on fait une distinction de cas selon l'opération d'édition. □

Démonstration de la proposition 7.1. C'est une conséquence directe du fait que $\mathbf{lev}(\epsilon,\epsilon) = 0$ et du lemme précédent en prenant $a = x[i], b = y[j], u = x[0...i-1]$ et $v = [y....j-1]$. □

Corollaire 7.1 *L'algorithme DynamicLev retourne* $\mathbf{lev}(x,y)$ *sur l'entrée* (x,y).

Complexité.

Proposition 7.2 *L'algorithme* `DynamicLev`, *sur une entrée* (x,y), *s'exécute en temps* $\mathcal{O}(|x| \times |y|)$ *et en espace* $\mathcal{O}(\min(|x|,|y|))$.

Démonstration. La complexité en temps s'obtient directement via la double boucle. Pour la complexité en espace, il suffit de remarquer que seulement deux lignes (ou deux colonnes) sont nécessaires simultanément. □

Développement 8

Automate des motifs

Auteur·e·s: Marin Malory
Références : [Lesesvre et al., 2020]

Ce développement présente un algorithme permettant de résoudre le problème de recherche de motif dans un texte. Pour cela, on construit un automate minimal en pré-traitement, permettant ensuite de résoudre le problème linéairement en la taille du texte. Ainsi, il s'intègre aussi bien dans la leçon 9 que dans la leçon 29. Enfin, il peut illustrer la leçon 2 si la programmation orienté automate est abordée.

Introduction. Le problème de recherche d'un motif M dans un texte T consiste à déterminer si M apparaît comme facteur (sous-mot) de T. On pose $T = t_1...t_n$ et $M = m_1...m_k$, et pour $1 \leq i \leq k$, on pose $M_i = m_1...m_i$ le préfixe de M de longueur i. On pose aussi par convention $M_0 = \epsilon$.

Si $u, v \in \Sigma^*$, on note $u \sqsubset v$ si u est suffixe de v. On note aussi

$$\sigma(u) = \max\{i : M_i \sqsubset u\}$$

c'est-à-dire la taille du plus grand préfixe de M qui est également suffixe de u.

L'objectif est alors de construire un algorithme résolvant le problème en $\mathcal{O}(n)$ avec un pré-traitement polynomial en k.

Construction d'un automate. Soit $A = (Q, \Sigma, I, F, \delta)$ l'automate déterministe complet défini par :

- $Q = \{0, ..., k\}$;
- $I = \{0\}$;
- $F = \{k\}$;
- pour tout $q \in Q$ et $a \in \Sigma$, $\delta(q, a) = \sigma(M_q a)$.

On va alors montrer que le langage reconnu par notre automate est $\Sigma^* M$, ce qui nous permettra de l'utiliser afin de résoudre le problème initial. On montrera ensuite que cet automate est minimal, et qu'il n'y a donc pas besoin de le minimiser avant de traiter le texte.

Proposition 8.1 *Pour tout mot $u \in \Sigma^*$, on a $\delta^*(0, u) = \sigma(u)$. Ainsi, $L(A) = \Sigma^* M$.*

Un premier lemme. Ce premier lemme, manipulant préfixe et suffixe, nous sera utile pour montrer la proposition précédente.

Lemme 8.1 *Soient $u, v \in \Sigma^*$ et $a \in \Sigma$.*

1. *Si $u \sqsubset v$, alors $\sigma(u) \leq \sigma(v)$.*
2. *$\sigma(ua) \leq \sigma(u) + 1$.*
3. *$\sigma(ua) = \sigma(M_{\sigma(u)}a)$*

La démonstration de ce lemme est difficile à faire en direct au tableau. Il sera judicieux de justifier la démonstration avec des dessins, quitte à le faire manière un peu moins formel pour gagner en clarté.

Remarque 8.1 *Si $M_i \sqsubset u$ $(0 \leq i \leq k)$, alors $i \leq \sigma(u)$. De plus, on a $\sigma(M_i) = i$.*

Démonstration.

1. Si $u \sqsubset v$, alors tout suffixe de u est suffixe de v. Ainsi, on a $\{i : M_i \sqsubset u\} \subset \{i : M_i \sqsubset v\}$, et donc $\sigma(u) \leq \sigma(v)$.
2. Si $\sigma(ua) = 0$, alors puisque $\sigma(u) \geq 0$, l'inégalité est vérifiée. Sinon, on a $\sigma(ua) - 1 \geq 0$. On remarque alors que $M_{\sigma(ua)-1} \sqsubset u$, et ainsi, $\sigma(ua) - 1 \leq \sigma(u)$.
3. On montre le résultat par double inégalité.
   - Par définition $M_{\sigma(u)} \sqsubset u$ et donc $M_{\sigma(u)}a \sqsubset ua$ et donc d'après le point 1, $\sigma(M_{\sigma(u)a}) \leq \sigma(ua)$.
   - Pour l'autre inégalité, on observe que $M_{\sigma(u)}a$ et $M_{\sigma(ua)}$ sont deux suffixes de ua. Ainsi, le plus court des deux mots est suffixe de l'autre. D'après le point 2, on a $|M_{\sigma(ua)}| = \sigma(ua) \leq \sigma(u) + 1 = |M_{\sigma(u)}a|$. Ainsi, $M_{\sigma(ua)} \sqsubset M_{\sigma(u)}a$, d'où $\sigma(ua) = \sigma(M_{\sigma(ua)}) \leq \sigma(M_{\sigma(u)}a)$ d'après le point 1.

□

Nous pouvons maintenant revenir au langage reconnu par l'automate défini ci-dessus.

Démonstration. On procède par récurrence sur la longueur l de u.

- Si $l = 0$, alors $u = \epsilon$ et $\delta^*(0, \epsilon) = 0 = \sigma(\epsilon)$.
- Supposons que $l > 0$ et que pour tout mot $v \in \Sigma^{l-1}$ vérifie $\delta^*(0, v) = \sigma(v)$. Soit $u \in \Sigma^l$, qu'on écrit $v = ua$ avec $|u| = l - 1$ et $a \in \Sigma$. En appliquant l'hypothèse de récurrence :

$$\delta^*(0, u) = \delta(\delta^*(0, v), a) = \delta(\sigma(v), a) = \sigma\left(M_{\sigma(v)}a\right) = \sigma(va) = \sigma(u)$$

Cela conclut la récurrence.

Enfin, par définition de σ, $\sigma(u) = k$ si et seulement si $M \sqsubset u$. Ainsi,

$$u \in L(A) \Leftrightarrow \delta^*(0, u) = k \Leftrightarrow \sigma(u) = k \Leftrightarrow M \sqsubset u \Leftrightarrow u \in \Sigma^* M$$

et donc $L(A) = \Sigma^* M$. □

Enfin, montrons que cet automate est minimal au sens de Nérode.

Lemme 8.2 *Pour $i, j \in Q$ tels que $i > j$, le mot $m_{i+1}...m_k$ est accepté par A à partir de i mais pas de j. Ainsi, A est minimal.*

Démonstration. On note $N_i = m_{i+1}...m_k$ le suffixe de M de taille $k-i$. Par construction, on a $\delta^*(0, M_i) = i$, d'où

$$\delta^*(i, N_i) = \delta^*(0, M_i N_i) = \delta^*(0, M) = k$$

Ainsi, N_i est accepté à partir de i dans A.

Pour $j < i$, alors $|M_j N_i| = j + k - i < k = |M|$. Ainsi, M n'est pas un suffixe de $M_j N_i$ et donc

$$\delta^*(j, N_i) = \delta^*(0, M_j N_i) = \sigma(M_j N_i) < k$$

donc N_i n'est pas accepté à partir de j dans A.

On en déduit de même que si $i > j$, les états i et j de A ne sont pas équivalents pour l'équivalence de Nérode. Comme A est déterministe et complet, ceci assure que A est minimal. □

Algorithme. La construction de A (et plus particulièrement de sa fonction de transition δ) n'utilise que le motif M et non le texte T. On peut représenter cette fonction via un tableau bidimensionnelle de taille $(k+1) \times |\Sigma|$. On peut alors remplir cette table avec les différentes valeurs de σ, ce qui se fait en temps polynomial en k. Enfin, on lit le texte T lettre par lettre et si on atteint l'état k, on a trouvé une occurrence de du motif M. Si on atteint la fin de T sans jamais atteindre k, alors M n'apparaît pas dans T.

Développement 9

Algorithme d'approximation pour un problème de routage

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Dans ce développement, on étudie un problème de routage qui s'avère être **NP***-complet, et on propose d'en étudier un algorithme d'approximation. Dans ce cadre, ce développement rentre dans la leçon 11, mais aussi dans la leçon 26 si celle-ci traite des algorithmes d'approximations. Puisque le problème étudié concerne du routage, il s'intègre notamment dans la leçon 21 si un point de vue théorique est abordé.*
Enfin, d'une manière plus large, ce problème consiste à trouver des chemins disjoints dans un graphe, donnant une application aux algorithmes de plus courts chemins présentés en leçon 7.

Introduction. Le routage consiste à envoyer un paquet d'une source à une destination dans un réseau. Plus formellement, on représente ce réseau comme un graphe orienté $G = (V, E)$, et on définit un ensemble de requêtes $\mathcal{R} \subset V \times V$ que l'on essaye de satisfaire.

Dans ce problème, tous les paquets d'une même requête passent par le même chemin dans le réseau. De plus, deux chemins de deux requêtes différentes ne peuvent pas partager une arête. Autrement dit, chaque requête réserve un chemin qu'elle seule peut utiliser. On veut alors maximiser le nombre de requêtes satisfaites.

Définition 9.1 (Maximum Edge-Disjoint Paths (MEDP))
Entrée :
— *Graphe* $G = (V, E)$
— *Ensemble de requêtes* $\mathcal{R} \in \mathcal{P}(V \times V)$
Sortie :

$$\max_{A \subset \mathcal{R} \text{ est réalisable}} |A|$$

où $A \subset \mathcal{R}$ *est réalisable si pour tout* $r_i = (s_i, t_i) \in \mathcal{R}$ *il existe un chemin* π_i *de* s_i *à* t_i *dans* G*, et tel que pour tout* $(R_i, R_j) \in A^2$*, si* $i \neq j$ *alors* π_i *et* π_j *ne partagent aucune arête.*
On notera A^* *une solution optimale à ce problème.*

Théorème 9.1 (Admis) **MEDP** *est NP-complet.*

Algorithme. On présente ici un algorithme glouton plutôt naïf, nommé *Shorts-Requests-First* ou SRF. L'idée est de router les requêtes courtes d'abord, c'est-à-dire celles dont le plus court chemin est minimal

par rapport aux autres requêtes disponible.

Algorithme 9.1 : Short-Requests-First(G,$\mathcal{R}$)

$A \leftarrow \emptyset$;
tant que *il existe une requête de $\mathcal{R}$ qui peut être router* **faire**
  choisir : $r = (s, y) \leftarrow$ une requête de $\mathcal{R}$ qui a le plus petit plus court chemin ;
  accepter : $A \leftarrow A \cup \{R_i\}$;
  router : $\pi \leftarrow$ un plus court chemin de s_i à t_i dans G ;
  supprimer : $\mathcal{R} \leftarrow \mathcal{R} \backslash \{r\}$
  élaguer : enlever les arêtes de π_i de G

Complexité. En utilisant l'algorithme de Floyd-Warshall pour la partie « choisir », on peut calculer l'ensemble des plus courts chemins allant d'une source s à une destination t avec une complexité de $\mathcal{O}(|V|^3)$. La complexité totale de l'algorithme est alors un $\mathcal{O}(|\mathcal{R}|.|V|^3)$.

Théorème 9.2 *L'algorithme 9.1 a un ratio d'approximation de $\lceil\sqrt{m}\rceil$ pour le problème* **MEDP** *sur un graphe avec m arêtes, et cette borne est atteinte.*

Démonstration. On commence par montrer que la borne est atteinte. Soit $q \geq 2$.

On construit un graphe $G = (V, E)$, avec $V = \{u_{10}\} \cup \{u_{ij} | 1 \leq i, j \leq q\}$. On choisit les arêtes suivantes :

— Un premier chemin p_0 qui est une chaîne de $q + 1$ sommets :

$$u_{10} \rightarrow u_{11} \rightarrow ... \rightarrow u_{1q}$$

— Pour $1 \leq i \leq q$, on construit le chemin :

$$u_{1i} \rightarrow u_{1i} \rightarrow ... \rightarrow u_{qi}$$

On remarque alors que $m = |E| = q^2$. L'ensemble des requêtes est défini par :

$$\mathcal{R} = (u_{10}, u_{1q}) \cup \{(u_{1(i-1)}, u_{1q}) | 1 \leq i \leq n\}$$

c'est-à-dire l'ensemble des couples (début,fin) des $q + 1$ chemins du graphe G. Puisque tous les chemins ont la même longueur q, l'algorithme SRF va choisir la requête $R_0 = (u_{10}, u_{1q})$ en premier. Or chaque requête suivante partage une arête avec le seul chemin possible pour satisfaire cette requête, donc $|A| = 1$. Or, la solution optimale consiste à choisir les q autres requêtes, donc $|A^*| = q = \lceil\sqrt{m}\rceil$.

Cas général :

Étant donné une instance de **MEDP** $(G, \mathcal{R})$ où G a m arrêtes, on note $A^* = \{r_i^*\}$ une solution optimale. On note p_i^* le chemin associé à $r_i^* \in A^*$.

L'idée consiste à procéder par stratégie adverse : on lance l'algorithme et en parallèle, on supprime des requêtes de A^*.

On lance SRF sur $(G, \mathcal{R})$. À chaque itération, SRF sélectionne une requête $r \in \mathcal{R}$ de chemin π.

— Si $r \in A^*$, on supprime r de A^*.

— Pour tout $r^* \in A^*$ telle que $\pi^* \cap \pi \neq \emptyset$, on supprime r^* de A^*.

Lemme 9.1 *Pour montrer que $|A^*| \leq \lambda|A|$ pour une certaine constante λ, il suffit de montrer que lorsque l'algorithme SRF accepte une requête, pas plus de λ requêtes sont supprimées de A^*.*

Démonstration. L'algorithme accepte $|A|$ requêtes (autant d'itérations), et à la fin, A^* est vide. □

Si SRF sélectionne $r \in \mathcal{R}$ de chemin π. Il y a deux cas :

1. Si π_i a au plus $\lceil \sqrt{m} \rceil - 1$ arêtes. Il y a au plus $\lceil \sqrt{m} \rceil - 1$ requêtes de A^* dont le chemin partage une arête avec π_i. En ajoutant la possibilité de supprimer R_i lui-même de A^*, on supprime au maximum $\lceil \sqrt{m} \rceil$ requêtes de A^*.
2. Si π_i a au moins $\lceil \sqrt{m} \rceil$ arêtes. Alors toutes les requêtes qui sont dans A^* ont un chemin de longueur au moins $\lceil \sqrt{m} \rceil$, parce que sinon elles auraient été choisies par l'algorithme SRF (on choisit le plus petit des plus petits chemins).
   Ainsi, toutes les requêtes restantes de A^* ont un chemin associé de longueur $\lceil \sqrt{m} \rceil$ et ont toutes des chemins disjoints deux à deux (y compris R_i si $R_i \in A$). Ainsi, par l'absurde, si ce nombre de requêtes restantes est strictement supérieur à $\lceil \sqrt{m} \rceil$, alors on obtient en tout strictement plus de m arêtes dans le graphe, c'est absurde. Il y a donc moins de $\lceil \sqrt{m} \rceil$ requêtes dans A^*, et donc moins de $\lceil \sqrt{m} \rceil$ requêtes qui peuvent être supprimées.

Ainsi, d'après le lemme, on a pour toute instance de $\mathbf{MEDP}$:

$$|A^*| \leq \lceil \sqrt{m} \rceil \times |A|$$

□

Le ratio $\lceil \sqrt{m} \rceil$ du théorème 9.2 est optimal, on ne peut pas faire mieux à moins que $\mathbf{P} = \mathbf{NP}$.

Développement 10

Introduction à la méthode probabiliste

Auteur·e·s: Marin Malory
Références : [Mitzenmacher and Upfal, 2005]

Ce développement présente une introduction à la méthode probabiliste. Cette méthode permet très souvent de donner des algorithmes d'approximations probabilistes et s'insère donc dans la leçon 11. Si les algorithmes d'approximations sont mentionnés dans la leçon 26, ce développement permet d'illustrer une manière d'obtenir des certificats d'existence de solution à des problèmes d'optimisation.

Méthode probabiliste. La méthode probabiliste est un moyen de montrer l'existence d'objets par un simple résonnement de dénombrement. Si la probabilité d'obtenir un objet ayant certaine propriété est strictement positive, alors un tel objet existe. Par exemple, si la probabilité de tirer au sort un ticket gagnant est non nulle, alors il y a au moins un ticket gagnant.

Cette méthode est non-constructiviste en générale, on montre juste l'existence sans vraiment donner l'objet. Mais le plus souvent, une preuve utilisant la méthode probabiliste est équivalente à un algorithme probabiliste permettant de construire cet objet.

Argument d'espérance. On utilisera le lemme suivant :

Lemme 10.1 *On considère un univers Ω et X une variable aléatoire sur Ω. Si $\mathbb{E}(X) = \mu$, alors $P(X \geq \mu) \geq 0$ et $P(X \leq \mu) > 0$.*

Ainsi, il existe au moins une réalisation de X supérieure ou égal à μ, et au moins une réalisation de X inférieure ou égal à μ.

Application : MAX-SAT.

Théorème 10.1 *Étant donné un ensemble de m clauses, on note k_i le nombre de littéraux dans la $i$ème clause pour $1 \leq i \leq m$. Soit $k = \min_{1 \leq i \leq m} k_i$. Il existe une distribution de valeurs de vérités qui satisfait au moins*

$$\sum_{i=1}^{m} (1 - 2^{-k_i}) \geq m(1 - 2^{-k})$$

clauses.

Démonstration. Pour chaque variable, on tire au sort sa valeur de vérité. En notant X_i pour $1 \leq i \leq m$ la variable aléatoire qui vaut 1 la $i$ème clause est satisfaite et 0 sinon, on a $P(X_i = 0) = 2^{-k_i}$ par indépendance des choix. En notant N le nombre de clause satisfaite, on a $N = \sum_{i=1}^{n} X_i$ et par linéarité

de l'espérance :

$$\mathbb{E}(N) = \sum_{i=1}^{n} \mathbb{E}(X_i) = \sum_{i=1}^{m} (1 - 2^{-k_i})$$

Par le lemme ci-dessus, on peut conclure. □

Ainsi, la preuve ci-dessus peut être transformer en un algorithme qui assigne à chaque variable une valeur de vérité au hasard. Son temps d'exécution est un $\mathcal{O}(m)$, et il renvoie une distribution qui satisfait en espérance $m(1 - 2^{-k})$ clauses. Cet algorithme est donc une $1 - 2^{-k}$-approximation randomisé.

Application : Ensemble indépendant.

Théorème 10.2 *Soit $G = (S, A)$ un graphe connexe à n sommets et $m \geq n/2$ arêtes. Alors G contient un ensemble indépendant ayant au moins $n^2/4m$ sommets.*

Démonstration. Soit $d = 2m/n \geq 1$ le degré moyen d'un sommet dans G. On considère l'algorithme probabiliste suivant :

Algorithme 10.1 : EnsembleIndépendantRandomisé(G)

Supprimer chaque sommet de G (avec ses arêtes adjacentes) de manière indépendante avec une probabilité $1 - 1/d$;
Pour chaque arête restante, la supprimer avec un des sommets adjacent pris au hasard.

L'algorithme renvoie un ensemble indépendant puisque chaque arête a été supprimée.

Remarque : l'algorithme se passe en deux phases, une phase d'échantillonnage, et une phase de modification. On parle de méthode *Sample and Modify*.

Soit X le nombre de sommets qui survivent à la première étape de l'algorithme. X suit une loi binomiale de paramètre n et $1/d$. Ainsi, $\mathbb{E}(X) = n/d$.

Soit Y le nombre d'arêtes qui survivent à la seconde étape. Chaque arête survit si, et seulement si ses deux sommets adjacents survivent, donc Y est la somme de m variables de Bernoulli de paramètre $(1/d)^2$. Ainsi,

$$\mathbb{E}(Y) = m\left(\frac{1}{d}\right) = \frac{nd}{2d^2} = \frac{n}{2d}$$

La seconde étape retire toutes les arêtes restantes et donc au plus Y sommets. Ainsi, lorsque l'algorithme termine, le nombre de sommets restants est $X - Y$, et

$$\mathbb{E}(X - Y) = \frac{n}{d} - \frac{n}{2d} = \frac{n}{2d}$$

Ce qui conclut notre preuve. □

L'algorithme ci-dessus nous donne donc une $\max(n/4m, 4m/n)$ approximation randomisée, ainsi qu'une borne inférieure sur le cardinal du plus grand ensemble indépendant, noté α.

Développement 11

Algorithme CYK

Auteur·e·s: Marin Malory
Références : [Lesesvre et al., 2020]

Ce développement présente l'algorithme Cocke-Younger-Kasami qui résout le problème du mot pour des grammaires sous forme normale de Chomsky. Il s'agit d'un algorithme de programmation dynamique, s'intégrant ainsi dans la leçon 12, qui permet de réaliser une analyse syntaxique, s'insérant dans la leçon **??**.

Introduction. L'algorithme CYK permet de résoudre le problème du mot pour les grammaires algébriques. On dit qu'une grammaire $G = (V, \Sigma, R, S)$ est sous forme normale de Chomsky si elle ne contient que des règles de la forme $A \to a$ (avec $a \in \Sigma$ et $A \in V$) ou $A \to A_1 A_2$ ($A_1, A_2 \in V$).

Définition 11.1 (Problème de mot) *Étant donné un mot $w = w_1 ... w_n$ et une grammaire G sous forme normale de Chomsky, a-t-on $w \in L(G)$?*

Algorithme CYK. On utilisera de la programmation dynamique. Pour tous les indices $1 \leq i \leq j \leq n$, on note

$$E_{i,j} = \{A \in V, A \to^* w_i ... w_j\}$$

Ainsi, $w \in L(G)$ si, et seulement si $S \in E_{1,n}$. On va donc calculer $E_{1,n}$ par programmation dynamique.

Initialisation. Montrons que pour tout $1 \leq i \leq n$, on a

$$E_{i,i} = \{A \in V : A \to w_i \in R\}$$

En effet, si $A \in E_{i,i}$, alors $A \to^* w_i$. Or, puisque G est sous forme normale de Chomsky, on n'a jamais $X \to^* \epsilon$ pour $X \in V$. Ainsi, la première règle de $A \to^* w_i$ n'est pas de la forme $A \to A_1 A_2$ qui donnerait un mot d'au moins deux lettres. On en déduit qu'on applique une règle de la forme $A \to a$, et donc $a = w_i$. Ainsi, $A \to w_i \in R$.

L'autre inclusion est triviale.

Récurrence. Soit $1 \leq i < j \leq n$, montrons

$$E_{i,j} = \bigcup_{k=i}^{j-1} \bigcup_{\substack{B \in E_{i,k} \\ C \in E_{k+1,j}}} \{A \in V : A \to BC \in R\}$$

Montrons l'inclusion indirecte. Soit $A \in V$ tel qu'il existe k tel que $i \leq k \leq j-1$, $B \in E_{i,k}$ et $C \in E_{k+1,j}$ avec $A \to BC \in R$. On a alors $B \to^* w_i ....w_k$, et $C \to^* w_{k+1} ... w_j$. Ainsi,

$$A \to BC \to^* w_i ....w_k C \to^* w_i ....w_j$$

et donc $A \in E_{i,j}$.

Réciproquement, soit $A \in E_{i,j}$, c'est-à-dire $A \to^* w_i...w_j$. On a au moins deux lettres dans le mot $w_i...w_j$, la première règle appliquée est donc de la forme $A \to BC$. On utilise alors l'arbre de dérivation de $A \to^* w_i...w_j$ dans lequel A est la racine et a deux enfants, B et C. Pour exhiber k, il suffit de prendre le nombre l de feuilles dans l'arbre enraciner en B, et on pose $k = i + l - 1$. Ainsi, on a $B \to^* w_i...w_k$ et $C \to^* w_{k+1}...w_j$. Cela conclut la preuve.

Algorithme. L'algorithme va donc d'abord calculer les $A_{i,i}$ et remonter jusqu'à l'ensemble $E_{1,n}$ en calculant diagonale par diagonale.

Algorithme 11.1 : CYK(w,G)

```
pour 1 ≤ i ≤ j ≤ n faire
    E_{i,j} ← ∅ ;
pour i = 1...n faire
    pour A → a ∈ R faire
        si a = w_i alors
            A_{i,i} ← E_{i,i} ∪ {A} ;
pour d = 2...n faire
    pour (i, j) sur la d-diagonale supérieure faire
        pour k = i...j − 1 faire
            pour A → B_1 B_2 faire
                si B_1 ∈ E_{i,k} et B_2 ∈ E_{k+1,j} alors
                    E_{i,j} → E_{i,j} ∪ {A} ;
retourner S ∈ E_{1,n} ;
```

Implémentation et complexité. On veut pouvoir ajouter et vérifier rapidement si des non-terminaux appartiennent à un ensemble $E_{i,j}$. On peut alors, pour tout $1 \leq i \leq j \leq n$, utiliser un tableau booléen de taille $|V|$ pour avoir les deux opérations en temps constant. On a alors une complexité temporelle en $\mathcal{O}(|R| \times n^3)$. Pour la complexité spatiale, on a besoin de $\frac{n(n+1)}{2}$ tableaux de taille $|V|$, donc un $\mathcal{O}(n^2|V|)$.

Commentaires. Il faut d'abord transformer une grammaire pour la mettre sous forme normale de Chomsky. Cette transformation peut faire exploser la taille de la grammaire. Il faut aussi vérifier qu'on a bien $\epsilon \notin L(G)$, ce qui peut se faire par saturation.

Développement 12

Calcul de premier en analyse lexicale

Auteur·e·s: Marin Malory
Références : [Lesesvre et al., 2020]

Ce développement propose de montrer l'équivalence entre la définition descriptive et axiomatique de l'ensemble **premier** *en analyse syntaxique. Cette nouvelle définition permet notamment de calculer cet ensemble par saturation. Ainsi, ce développement s'insère dans la leçon 25.*

Définition de premier. On rappelle ici la définition descriptive de l'ensemble **premier**. On note ϵ_0 un symbole frais qui peut être interprété comme « première lettre de ϵ ».

Définition 12.1 *Soit G une grammaire, et $w \in (V \cup \Sigma)^*$, on définit :*

$$\mathbf{premier}(w) = \begin{cases} \mathbf{premier}'(w) & \text{si } w \not\Rightarrow^* \epsilon \\ \mathbf{premier}'(w) \cup \{\epsilon_0\} & \text{si } w \Rightarrow^* \epsilon \end{cases}$$

où :

$$\mathbf{premier}'(w) = \{a \in \Sigma | w \Rightarrow^* aw', w' \in (V \cup \Sigma)^*\}$$

Définition axiomatique. On considère une vision ensembliste des fonctions. On veut montrer que la fonction **premier** est la plus petite partie P de $(V \cup \Sigma)^* \times (\Sigma \cup \{\epsilon_0\})$ telle que

1. $a \in P(a)$;
2. $\epsilon_0 \in P(\epsilon)$;
3. si $N \to w_1...w_n$, alors $P(w_1....w_n) \subset P(N)$;
4. si $N \to \epsilon$, alors $\epsilon_0 \in P(N)$;
5. $(P(w_1) \backslash \{\epsilon_0\}) \subset P(w_1....w_n)$;
6. si $\epsilon_0 \in P(w_1)$, alors $P(w_2...w_n) \subset P(w_1....w_n)$

Lemme 12.1 *La fonction* **premier** *vérifie les axiomes 1 à 6.*

Démonstration.

1. Pour $a \in \Sigma$, on a $a \Rightarrow^* a$ et donc $a \in \mathbf{premier}(a)$.
2. De même, $\epsilon \Rightarrow^* \epsilon$, et donc $\epsilon_0 \in \mathbf{premier}(\epsilon)$;
3. Si $N \to w_1...w_n$. Soit $a \in \mathbf{premier}(w_1...w_n)$.
   Si $a = \epsilon_0$, alors $w_1...w_n \Rightarrow^* \epsilon$, et donc par transitivité $N \Rightarrow^* \epsilon$. Ainsi, $\epsilon_0 \in \mathbf{premier}(N)$.
   Si $a \neq \epsilon_0$, on a $w_1...w_n \Rightarrow aw'$, et donc encore par transitivité, $N \Rightarrow^* aw'$, et finalement $a \in \mathbf{premier}(N)$.

4. Si $N \Rightarrow \epsilon$, alors $N \Rightarrow^* \epsilon$ et donc $\epsilon_0 \in \mathbf{premier}(N)$.
5. Soit $w_1...w_n \in (\Sigma \cup V)^*$. Soit $a \in \mathbf{premier}(w_1)\backslash\{\epsilon_0\}$. Ainsi, $a \in \mathbf{premier}'(w_1)$, et donc $w_1 \Rightarrow^* aw_1'$. En appliquant les mêmes règles au mot $w_1....w_n$, on a $w_1....w_n \Rightarrow^* aw_1'w_2...w_n$, et donc $a \in \mathbf{premier}(w_1...w_n)$.
6. Soit $w_1...w_n \in (\Sigma \cup V)^*$, avec $\epsilon_0 \in \mathbf{premier}(w_1)$. Alors, on a $w_1 \Rightarrow^* \epsilon$. En appliquant les même règle, on a $w_1...w_n \Rightarrow^* \epsilon w_2...w_n = w_2...w_n$. On en déduit directement $\mathbf{premier}(w_2...w_n) \subset \mathbf{premier}(w_1...w_n)$.

□

Lemme 12.2 *Soit P la plus petite partie vérifiant les axiomes 1 à 6. Montrer que pour tout $n \in \mathbb{N}$, pour tout $a \in \Sigma$, pour tout $w, w' \in (\Sigma \cup V)^*$, on a*

— *si $w \Rightarrow^n aw'$, alors $a \in P(w)$;*

— *si $w \Rightarrow^n \epsilon$, alors $\epsilon_0 \in P(w)$.*

En admettant ce lemme pour l'instant, on peut montrer le résultat voulu.

Théorème 12.1 $\mathbf{premier}$ *est la plus petite partie P vérifiant les propriétés 1 à 6.*

Démonstration. Soit P la plus petite partie vérifiant les axiomes 1 à 6. Puisque $\mathbf{premier}$ vérifie ces axiomes d'après le lemme 12.1, on a $P \subset \mathbf{premier}$. Le lemme 12.2 nous permet de montrer l'autre inclusion. En effet, si $a \in \mathbf{premier}(w)$, il existe $w' \in (\Sigma \cup V)^*$ tel que $w \Rightarrow^* aw'$ (avec possiblement $a = \epsilon_0$ et donc $w' = \epsilon$). Par le lemme 12.2, on a directement $a \in P(w)$, et donc pour tout w, on a $\mathbf{premier}(w) \subset P(w)$.
Finalement, $\mathbf{premier} = P$. □

Algorithme. Cette définition axiomatique nous donne un moyen de calculer $\mathbf{premier}$ par saturation via l'algorithme 12.1.

Algorithme 12.1 : Calcul_Premier(w)

```
pour a ∈ Σ faire
    premier(a) ← {a} ;
premier(ε) ← {ε_0} ;
pour N → ε ∈ R faire
    premier(N) ← {ε_0} ;
tant que premier est modifié faire
    pour N → w_1...w_n ∈ R faire
        premier(N) ← premier(N) ∪ premier(w_1...w_n) ;
        si ε_0 ∉ premier(w_1) alors
            premier(w_1...w_n) ← premier(w_1...w_n) ∪ (premier(w_1)\{ε_0}) ;
        sinon
            premier(w_1...w_n) ← premier(w_1...w_n) ∪ premier(w_2...w_n) ;
```

Démonstration du lemme 12.2. On montre le résultat par récurrence forte sur n.
Pour $n = 0$, soit $a \in \Sigma$, $w, w' \in (\Sigma \cup V)^*$.

— Si $w = aw'$, alors on a $a \in P(a)$ d'après 1, et en particulier $a \in P(a)\backslash\{\epsilon_0\}$. D'après 5, on a $a \in P(aw')$, et donc $a \in P(w)$.

— Si $w = \epsilon$, alors par 2, on a directement $\epsilon_0 \in P(w)$.

On suppose la propriété vraie pour tout $k < n$ pour un certain $n > 0$. soit $a \in \Sigma$, $w, w' \in (\Sigma \cup V)^*$. On note $w = w_1...w_m$.

— Si $w \Rightarrow^n aw'$. On veut montrer que $a \in P(w)$. On distingue deux cas
  — Si $w_1 \in \Sigma$, alors $w_1 = a$. Ainsi par 1 et 5, on a directement $a \in P(w)$.
  — Si $w_1 \notin \Sigma$, ie $w_1 = N \in V$. On distingue deux cas :
    — S'il existe $j < n$ tel que

$$Nw_2...w_m \Rightarrow^j w_2...w_m \Rightarrow^{n-j} aw'$$

On a alors $\epsilon_0 \in P(N)$ par HR au rang j, et $a \in P(w_2...w_m)$ par HR au rang $n - j$. De plus, par l'axiome 6, on a $P(w_2...w_m) \subset P(w_1...w_m)$ et donc $a \in P(w)$.

    — Sinon, la première règle appliquée à N est de la forme $N \Rightarrow x_1...x_k$. De plus, puisque $x_1...x_k \not\Rightarrow^* \epsilon$, on prend i l'indice minimal tel que $x_i \Rightarrow^* a$. Il existe alors $j < n$ tel que

$$Nw_2...w_n \Rightarrow^j x_i...x_kw_2...w_m \Rightarrow aw'$$

Par application successive de 6, on a

$$P(x_i...x_k) \subset P(x_1...x_k)$$

Or par HR, on a $a \in P(x_i) \subset P(x_i...x_k)$ par 5. En combinant, on a $a \in P(x_1...x_k)$, puis $a \in P(N)$ par 3, et finalement par 5, on a $a \in P(Nw_2...w_m)$, c'est-à-dire $a \in P(w)$.

— Si $w \Rightarrow^n \epsilon$, on recommence avec w qui commence par un non terminal.

□

Développement 13

Voyageur de commerce

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Ce développement étudie le problème du voyageur de commerce qui consiste à trouver un chemin de poids minimal passant par tous les sommets une et une seule fois dans un graphe complet pondéré. Il s'insère alors naturellement dans la leçon 7. Tout d'abord, on montre la **NP***-complétude du problème, illustrant la leçon 26. Ensuite, on montre un résultat d'inapproximabilité et on présente un algorithme d'approximation dans le cas où la fonction de poids du graphe vérifie l'inégalité triangulaire. Ainsi, ce développement s'intègre dans la leçon 11.*

Définition du problème et complexité.

Définition 13.1 (Voyageur de commerce (TSP)) *Étant donné un graphe complet $G = (V, E)$ et une fonction de coût $w : E \to \mathbb{N}$ et nu entier k, existe-t-il un cycle C passant par chaque sommet une et une seule fois, avec $\sum_{e \in C} w(e) \leq k$?*

On se propose de montrer le théorème suivant :

Théorème 13.1 **TSP** *est* **NP***-complet.*

Démonstration. **TSP** est trivialement dans **NP**, un certificat étant la donnée du cycle. Il suffit alors de vérifier si ce cycle passe bien une et une seule fois par chaque sommet et si la somme des poids est bien inférieure à k.

Pour montrer que **TSP** est **NP**-dure, on réduit **HC** (circuit hamiltonien) à **TSP**.
Soit $G = (V, E)$ une instance de **HC**. On construit alors une instance de **TSP** de la manière suivante :

— on pose $G' = (V, E')$ le graphe complet contenant les mêmes sommets que G ;
— on pose $w : E \to \{0, 1\}$ où $w(e) = \mathbf{1}\{e \notin E\}$, c'est-à-dire toute les arêtes de G ont un poids de 0, et 1 sinon.
— on prend $k = 0$.

Cette nouvelle instance de **TSP** a bien une taille polynomiale en la taille de l'instance de départ.

Si G admet un cycle hamiltonien C. On montre que C est une solution de **TSP**. En effet, C passe par chaque sommet une et une seule fois, et puisqu'il ne passe que par des arêtes de E, on a $\sum_{e \in C} w(e) = 0$.

Réciproquement, si **TSP** admet une solution C vérifiant $\sum_{e \in C} w(e) = 0$, alors pour tout $e \in C$, $w(e) = 0$ et donc $e \in E$. Ainsi, C est une solution de **HC**. □

Inaproximabilité. On s'intéresse maintenant au problème d'optimisation associé à $\mathbf{TSP}$. On peut montrer qu'il n'existe aucun algorithme d'approximation, à moins que $\mathbf{P} = \mathbf{NP}$.

Théorème 13.2 *Pour tout $\lambda \geq 1$, il n'existe pas de λ-approximation de $\mathbf{TSP}$, à moins que $\mathbf{P} = \mathbf{NP}$.*

Démonstration. Soit $\lambda \geq 1$. On procède par l'absurde, on suppose qu'il existe une λ-approximation pour $\mathbf{TSP}$, et on montre que l'on peut résoudre $\mathbf{HC}$ avec cet algorithme.

Soit $G = (V, E)$ une instance de $\mathbf{HC}$. On transforme cette instance en une instance de $\mathbf{TSP}$:

— $G' = (V, E')$ le graphe complet ayant les mêmes sommets que G.

— $w : E' \to \mathbb{N}$ où

$$w(e) = \begin{cases} 1 & \text{si } e \in E \\ \lambda n + 1 & \text{sinon} \end{cases}$$

On note C_{opt} la solution optimale de $\mathbf{TSP}$ pour ce problème, et on note c_{opt} le poids du cycle associé. De même, on pose C_{algo} la solution retournée par notre algorithme d'approximation, et c_{algo} le poids du cycle. Par hypothèse, on a :

$$c_{algo} \leq \lambda \times c_{opt}$$

On considère deux cas :

— Si $c_{algo} \geq \lambda n + 1$, alors on a $c_{opt} > n$. On en déduit que G n'admet pas de cycle hamiltonien, car si c'était le cas, ce cycle ne passerait que par des arêtes de poids 1 et aurait donc un poids total de n.

— Si $c_{algo} < \lambda n + 1$, alors on remarque que C_{algo} ne passe que par des arêtes de E, puisque s'il passait par une seule arête hors de E, son poids dépasserait $\lambda n + 1$. La solution c_{algo} est donc un cycle hamiltonien de G.

Ainsi, le résultat de notre algorithme nous permet de résoudre le problème $\mathbf{HC}$, ce qui conclut la preuve. □

Cas euclidien. On termine par une restriction du problème du voyageur de commerce, nous permettant de trouver un algorithme d'approximation. On définit $\mathbf{TSP} - \mathbf{EUCLIDIEN}$ le problème $\mathbf{TSP}$ dans lequel la fonction de poids w vérifie l'inégalité triangulaire :

$$\forall v_1, v_2, v_3 \in V, w(v_1, v3) \leq w(v_1, v_2) + w(v_2, v_3)$$

On rappelle que le graphe est complet.

On définit l'algorithme suivant :

Algorithme 13.1 : Spanning-tsp(G,w)

$T \leftarrow$ arbre couvrant de poids minimum de G ;
$e_1, ..., e_n \leftarrow$ Parcours-Profondeur(T) ;
$C \leftarrow \{(v_i, v_{i+1})\}_{1 \leq i < n} \cup \{(v_n, v_1)\}$;
retourner C

Théorème 13.3 *L'algorithme Spanning-tsp est une 2-approximation pour le problème $\mathbf{TSP} - \mathbf{EUCLIDIEN}$.*

Démonstration. Soit $G = (V, E)$ et w une instance de $\mathbf{TSP} - \mathbf{EUCLIDIEN}$. On note c_{opt} le coût optimal pour le problème. On note $w(T)$ la somme des poids des arêtes de l'arbre couvrant de poids minimal retenu par l'algorithme.

En prenant le cycle optimal, si on enlève une arête, on obtient un arbre. Ainsi, on a $c_{opt} \geq w(T)$.

On considère maintenant la solution C_{algo} de poids c_{algo} retournée par l'algorithme. On note O l'ordre du parcours et ainsi, C est obtenu en gardant seulement la première occurrence de chaque sommet dans O.

Remarque 1 : dans l'ordre O, on rencontre exactement 2 fois chaque arête de T.

Remarque 2 : C_{algo} est obtenu en supprimant de O des sommets. Or, en supprimant un sommet de O, on n'augmente pas le poids du chemin associé : $(x, y, z) \rightsquigarrow (x, z)$, on a $w(x, y) + w(x, z) \geq w(x, z)$.

Ainsi, en notant C_O le chemin associé à O, on a $c_{algo} = w(C_{algo}) \leq w(C_O)$ par la remarque 2. Or $w(C_0) = 2 \times w(T)$ par la remarque 1. Finalement, en combinant les inégalités,

$$c_{algo} \leq 2c_{opt}$$

ce qui conclut la preuve. □

Développement 14

2-SAT est linéaire

Auteur·e·s: Marin Malory
Références : [Lesesvre et al., 2020]

*Ce développement propose de montrer que le problème 2-**SAT** est décidable en temps linéaire. Pour cela, on montre que résoudre ce problème revient à déterminer les composantes fortement connexes d'un certain graphe. Ainsi, ce développement s'intègre dans la leçon 26 comme illustration non évidente d'un problème **P**. De plus, il s'insère naturellement dans la leçon 28 pour illustrer un problème de satisfiabilité. Enfin, ce développement illustre une application non tivial des algorithmes permettant de déterminer les composantes fortement connexes d'un graphe, tels que Tarjan ou Kosaraju, et s'intègre alors parfaitement dans la leçon 7.*

Définition 14.1 (2-SAT)
Données :
- *un ensemble de variables propositionnelles* $X = \{x_1, ..., x_n\}$
- *une formule* F *sous forme normale conjonctive où chaque clause est composée de* 2 *littéraux (un littéral étant une variable ou sa négation).*

Problème : *existe-t-il une valuation* $d : \{x_1, ..., x_n\} \to \{0, 1\}$ *telle que* $d(F) = 1$ *?*

On se propose de montrer le théorème suivant :

Théorème 14.1 *2-**SAT** est décidable en temps linéaire.*

Construction du graphe. Soit $X = \{x_1, ..., x_n\}$ et $F = C_1 \wedge C_2 \wedge ... \wedge C_m$ une instance de 2-**SAT**. On construit le graphe orienté $G = (V, E)$ avec :
- $S = \{x_1, \overline{x_1}, ..., x_n, \overline{x_n}\}$;
- pour $1 \leq i \leq m$, en notant $C_i = u \vee v$, on a $(\overline{u}, v) \in E$ et $(\overline{v}, u) \in E$.

Remarque 14.1 *On remarque que la taille du graphe est linéaire en la taille de la formule.*

Caractérisation de la satisfiabilité. On peut alors montrer le résultat suivant :

Théorème 14.2 *F est satisfiable si, et seulement si aucune composante fortement connexe de G ne contient à la fois une variable x et son complément $\overline{x}$.*

Lemme 14.1 *Soit d une valuation. On a $d(F) = 0$ si, et seulement s'il existe un chemin $l_1...l_k$ dans G avec $d(l_1) = 1$ mais $d(l_k) = 0$.*

Démonstration. Si $d(F) = 0$, alors il existe une clause $C = u \vee v$ telle que $d(C) = 0$. On prend alors le chemin $\overline{u}v$, et on a $d(\overline{u}) = 1$ et $d(v) = 0$.

Réciproquement, on suppose $d(F) = 1$. Montrons tout d'abord que si $uv \in E$ et $d(u) = 1$, alors $d(v) = 1$. En effet, si $uv \in E$, alors on a dans F la clause $\overline{u} \vee v$ ou la clause $v \vee \overline{u}$ (donc la même clause). On a directement $d(v) = 1$.

Ainsi, s'il existait un chemin $l_1...l_k$ dans G avec $d(l_1) = 1$, alors on a immédiatement $d(l_2) = ... = d(l_k) = 1$. □

Démonstration du sens direct du théorème 14.2. Par contraposée, on suppose qu'il existe x et $\overline{x}$ dans la même composante fortement connexe. Soit d une valuation, montrons $d(F) = 0$. Il y a deux cas :

- si $d(x) = 1$, alors puisqu'il existe un chemin $x \rightsquigarrow \overline{x}$ et $d(\overline{x}) = 0$, on conclut par le lemme 14.1 que $d(F) = 0$;
- si $d(x) = 0$, alors on a $d(\overline{x}) = 1$ et il existe un chemin $\overline{x} \rightsquigarrow x$ dans G. On conclut de la même manière.

Ainsi, F n'est pas satisfiable. □

Démonstration du sens indirect du théorème 14.2.

On suppose maintenant que pour tout $x \in X$, on a x et $\overline{x}$ dans des composantes fortement connexes différentes. On utilisera le lemme suivant.

Lemme 14.2 *Pour toute composante fortement connexe C de G, il existe une composante fortement connexe $\overline{C}$ tel que $u \in C \Leftrightarrow \overline{u} \in C$.*

Démonstration. Soit $u \in C$, on pose $\overline{C}$ la composante de $\overline{u}$. On montre alors que $v \in C \Leftrightarrow \overline{c} \in \overline{C}$.

On a $v \in C$ si, et seulement si $u \rightsquigarrow v$ et $v \rightsquigarrow u$. Or, s'il existe un chemin $u_1...u_k$ dans G, on a par définition de G l'existence du chemin $\overline{u_k}...\overline{u_1}$. Ainsi, on a $u \rightsquigarrow v$ si, et seulement si $\overline{v} \rightsquigarrow \overline{u}$. On peut conclure facilement. □

On peut maintenant construire la valuation. On considère le graphe des composantes fortement connexes de G. Il s'agit d'un DAG et on peut alors prendre un tri-topologique de ce graphe $C_1....C_n$. On construit la valuation d de la manière suivante : pour $i = 1...n$, si C_i n'a toujours pas été traitée, alors on pose $d(C_i) = 1$ et $d(\overline{C_i}) = 0$.

La fonction d est bien définie par hypothèse. Par l'absurde, si $d(F) = 0$, alors il existe un chemin $l_1....l_k$ dans G avec $d(l_1) = 1$ mais $d(l_k) = 0$. En particulier, il existe une arête ll' sur le chemin avec $d(l) = 1$ et $d(l') = 0$. On a l et l' dans deux CFC différentes C et C'. Puisque $d(C) = 1$, $\overline{C}$ est après dans le tri topologique ($\overline{C} < C$). Puisque $d(C') = 0$, $\overline{C'}$ a été traité avant et donc $C' < \overline{C'}$. Enfin, puisqu'on a une arête entre C et C', on a $C < C'$, et donc :

$$\overline{C} < C < C' < \overline{C'}$$

Or, puisque $ll' \in E$, on a $\overline{l'}\overline{l} \in E$, et donc $\overline{C'} < \overline{C}$, c'est absurde.

Finalement, F est satisfiable. □

Le théorème 14.1 est alors un corollaire du théorème 14.2 puisqu'il suffit de calculer les composantes fortement connexes en temps linéaire (via Tarjan ou Kosaraju), et de vérifier linéairement si une variable et sa négation sont dans la même CFC.

Développement 15

Théorème de compacité et application

Auteur·e·s: Marin Malory
Références : [Pinchinat et al., 2022]

L'idée de ce développement est de démontrer la théorème de compacité de la logique propositionnelle en partant du lemme de König, et de proposer une application en coloriage de graphe. Il s'intègre alors parfaitement dans la leçon 28. Si ce développent insiste plus sur le lemme de König, il peut alors illustrer les leçons 7 et 10.

Lemme de König. On se propose de montrer le lemme suivant. Pour un arbre T et un noeud x, on notera T_x le sous-arbre de T enraciné en x.

Lemme 15.1 (Lemme de König) *Tout arbre infini à branchement fini admet une branche infinie.*

Démonstration. Soit T un arbre enraciné ayant une infinité de nœuds et tel que chaque nœud interne a un degré fini. On construit une suite $(x_n)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, T_{x_n} est infini à branchement fini.

— On prend x_0 la racine de T, et la propriété est vérifiée au rang 0.

— On suppose qu'on dispose déjà de $x_0...x_n$. Par hypothèse de récurrence sur n, le sous-arbre de T_{x_n} est infini à branchement fini. On note $f_1, ..., f_k$ les enfants de cet arbre. Par l'absurde, si tous les (T_{f_i}) sont finis, chacun contenant n_i nœuds, alors T_{x_n} contient $\sum_{i=1}^{k} n_i$ noeuds, ce qui est absurde. Donc il existe $1 \leq i \leq k$ tel que T_{f_i} soit infini à branchement fini, on pose $x_{n+1} = f_i$.

On a ainsi construit par récurrence un chemin infini dans T. □

Un théorème de compacité.

Théorème 15.1 (Compacité de la logique propositionnelle) *Étant donné un ensemble dénombrable $\mathcal{F}$ de formules propositionnelles sur un ensemble de variables V dénombrable, $\mathcal{F}$ est satisfiable si, et seulement si, $\mathcal{F}$ est finiment satisfiable.*

Rappel :

1. $\mathcal{F}$ est satisfiable s'il existe une valuation $d : V \to \{0, 1\}$ telle que pour toute formule $F \in \mathcal{F}$, $d(F) = 1$.
2. $\mathcal{F}$ est finiment satisfiable si toute partie finie de $\mathcal{F}$ est satisfiable.

Démonstration. Le sens direct est trivial. Étant donné une valuation $d : \mathcal{F} \to \{0, 1\}$ telle que $d(\mathcal{F}) = 1$ et un sous-ensemble $\mathcal{G} \subset \mathcal{F}$ fini, on a toujours $d(\mathcal{G}) = 1$.

Réciproquement, on va construire un arbre afin d'appliquer le lemme de König. Soit $(\phi_n)_{n\in\mathbb{N}}$ une énumération de $\mathcal{F}$, et $(v_n)_{n\in\mathbb{N}}$ une énumération de V. On construit un arbre T de racine r de la manière suivante :

— pour $n \in \mathbb{N}$, les sommets de T de hauteur $n+1$ sont étiquetés par v_n ou $\overline{v_n}$;

— pour un sommet x de T de hauteur n, alors $y \in \{v_n, \overline{v_n}\}$ est un enfant de x si, et seulement s'il existe une valuation satisfaisant $x, y, \phi_0, ..., \phi_n$ et tous les ancêtres de x.

Montrons que T est infini à branchement fini. Chaque nœud a au plus deux enfants, donc T est à branchement fini. Il reste à montrer que T est infini, et pour cela, on montre que pour tout $n > 0$, T a au moins un nœud à la profondeur n. Soit $n > 0$, par hypothèse $\{\phi_0, ..., \phi_{n-1}\}$ est satisfiable par une valuation d. Ainsi, il existe au moins une branche de longueur n dans T en prenant pour $1 \leq i \leq n$, x_i si $d(x_i) = 1$ et $\overline{x_i}$ sinon.

Ainsi, par application du lemme, il existe une branche infini $(x_i)_{i>0}$ dans T. On pose alors $d(v_i) = 1$ si $x_i = v_i$ et $d(v_i) = 0$ si $x_i = \overline{x_i}$. Cette valuation satisfait alors toute formule de $\mathcal{F}$. □

Application : coloriage de graphe. On se propose de montrer le résultat suivant :

Théorème 15.2 *Soit $K > 0$ et G un graphe. G est K-coloriable si, et seulement si, tous ses graphes finis le sont.*

L'idée ici n'est pas de faire une preuve très détaillée du théorème. On pourra se contenter de donner les points clés de la preuve, l'idée principale étant de montrer une application du théorème de compacité.

Démonstration. Le sens direct est trivial, puisqu'il suffit de prendre une restriction du K-coloriage de G pour colorier un de ses sous-graphes.

Réciproquement, on définit pour tout graphe $H = (V, E)$ une formule Φ_H satisfiable si, et seulement si H est K-coloriable. L'idée est de considérer l'ensemble des variables $(x_{v,l})_{v\in V, l\in[\![1;K]\!]}$. On pourra prendre :

$$\Phi_H = \left(\bigwedge_{v\in V}\left(\bigvee_{1\leq l\leq K} x_{v,l}\right) \wedge \left(\bigwedge_{1\leq l<l'\leq K} \neg(x_{v,l} \wedge x_{v,l'})\right)\right) \wedge \left(\bigwedge_{uv\in E}\bigwedge_{1\leq l\leq K} \neg(x_{u,l} \wedge x_{v,l})\right)$$

□

Soit $\mathcal{H}$ l'ensemble des sous-graphes finis de G. On pose $\mathcal{F} = (\Phi_H)_{H\in\mathcal{H}}$ et on remarque que $\mathcal{F}$ est finiment satisfiable. En effet, si on prend un sous-ensemble fini de $\mathcal{F}$, l'union des sous-graphes finis correspondants est encore un sous-graphe fini de G et est donc K-coloriable. Ainsi, il existe une valuation satisfaisant ce sous-ensemble. Par le théorème de compacité, $\mathcal{F}$ est satisfiable par une valuation d de laquelle on peut extraire un K-coloriage de G.

Développement 16

Théorème de Myhill-Nérode

Auteur·e·s: Marin Malory
Références : [Belghiti et al., 2016]

Ce développement propose une preuve du théorème de Myhill-Nérode, ainsi qu'une application de ce théorème. L'application consiste à donner une condition nécessaire et suffisante pour qu'un langage soit rationnel dans le cas d'un alphabet unaire. Ce développement s'intègre dans la leçon 29.

Introduction. Étant donné un langage L et un mot u, on appelle résiduel de L par u l'ensemble

$$u^{-1}L = \{v \in \Sigma^* | uv \in L\}$$

Théorème 16.1 *Un langage est reconnaissable si, et seulement s'il n'admet qu'un nombre fini de résiduel.*

Sens direct. Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate complet déterministe reconnaissant L.

$$v \in u^{-1}L \Leftrightarrow uv \in L \Leftrightarrow \delta^*(q_0, uv) \in F \Leftrightarrow \delta^*(\delta^*(q_0, u), v) \in F$$

Pour $q \in Q$, on note $L_q = \{v \in \Sigma^* | \delta^*(q, v) \in F\}$, on a :

$$u^{-1}L = L_{\delta^*(q_0,u)}$$

Puisqu'il y a un nombre fini d'état, on obtient alors un nombre fini de résiduels (c'est-à-dire pour tout mot $u \in \Sigma^*, u^{-1}L \in \{L_q\}_{q\in Q}$).

Sens indirect. Soit Q l'ensemble des résiduels de L. On note $q_0 = L = \epsilon^{-1}L$, et F l'ensemble des résiduels contenant le mot vide. On pose $A = (\Sigma, Q, q_0, F, \delta)$ avec :

$$\delta(u^{-1}L, a) = (ua)^{-1}L$$

Montrons que $L(A) = L$. On a

$$\begin{aligned} w = w_1...w_n \in L(A) &\Leftrightarrow \delta^*(q_0, w) \in F \\ &\Leftrightarrow \epsilon \in \delta^*(\epsilon^{-1}L, w) \\ &\Leftrightarrow \epsilon \in \delta^*(w_1^{-1}L, w_2...w_n) \\ &\Leftrightarrow \epsilon \in \delta^*((w_1w_2)^{-1}L, w_3...w_n) \\ &\Leftrightarrow \epsilon \in w^{-1}L \\ &\Leftrightarrow w \in L \end{aligned}$$

Proposition 16.1 *Un automate minimal reconnaissant L contient autant d'état que L admet de résiduels.*

Démonstration. Étant donné un automate reconnaissant L, on peut associer à chaque état un résiduel via une fonction injective d'après la preuve du sens direct. Cet automate a donc plus d'états que L admet de résiduels. Enfin, d'après le sens indirect, cette borne est atteinte. □

Lemme 16.1 (de non pompage) *Soit L un langage sur un alphabet à une lettre $\Sigma = \{a\}$.*

L est rationnel si, et seulement si, pour tout $v \in \Sigma^*$, il existe des entiers $m > n > 0$ vérifiant $(v^m)^{-1}L = (v^n)^{-1}L$.

Démonstration. Montrons le sens direct. Si L est rationnel, alors il admet un nombre fini de résiduels. Ainsi, pour tout $v \in \Sigma^*$, d'après le principe des tiroirs, il existe $m > n > 0$ tels que $(v^m)^{-1}L = (v^n)^{-1}L$.

Montrons le sens indirect. En prenant $v = a$, notons $m > n > 0$ tel que $(a^m)^{-1}L = (a^m)^{-1}L$. Montrons, par récurrence forte sur k, que :

$$(a^k)^{-1}L \in \{(a^i)^{-1} | 0 \leq i \leq m-1\}$$

— La propriété est trivialement vraie pour $k < m$.

— On suppose la propriété vraie jusqu'au rang $k \geq m-1$. On a :

$$\begin{aligned}(a^{k+1})^{-1}L &= (a^{k+1-m})^{-1}((a^m)^{-1}L) \\ &= (a^{k+1-m})^{-1}((a^n)^{-1}L) \\ &= (a^{k+1-m+n})^{-1}L\end{aligned}$$

Or, $k+1-m+n \leq k$, on peut donc appliquer l'hypothèse de récurrence et conclure.

Ainsi, le nombre de résiduels de L est borné par m, et donc d'après le théorème de Myhill-Nérode, L est rationnel. □

Développement 17

Performance de l'algorithme des k-moyennes

Auteur·e·s: Marin Malory
Références : [Shalev-Shwartz and Ben-David, 2014]

Ce développement présente l'algorithme des k-moyennes, utilisé en apprentissage non supervisé. On présente aussi un résultat mathématiques de performance : on montre que la fonction objectif décroît à chaque itération de l'algorithme. Il s'insère dans la leçon 24.

Introduction. On considère un espace métrique $(\mathcal{X}, d)$ et un ensemble de points S. Une approche classique pour faire du clustering consiste à définir une fonction de coût parmi l'ensemble des clusterings (partitions de S) possibles et d'en trouver un de coût minimal. Cette fonction a pour entrée un clustering $C = (C_1, ..., C_k)$ et est à valeur dans $\mathbb{R}$.

On présente ici la fonction objectif la plus commune.

Fonction objectif des k-moyennes. Cette fonction objectif mesure la distance quadratique de chaque point de S au centroïde de son cluster. Étant donné un cluster C_i, son centroïde est défini par :

$$\mu_i(C_i) = \text{argmin}_{\mu \in \mathcal{X}} \sum_{x \in C_i} d(x, \mu)^2$$

On peut alors définir la fonction objectif :

$$G_{k-m}(C_1, ..., C_k) = \sum_{i=1}^{k} \sum_{x \in C_i} d(x, \mu_i(C_i))^2$$

Problème. Il est cependant complexe de trouver le minimum de cette fonction. En revanche, l'algorithme suivant est souvent utilisé. On considère la distance euclidienne $\|.\|$.

Algorithme 17.1 : k-Moyennes(k,X)

Données : $X \subset \mathcal{X}$; nombre de clusters k
$\mu_1, ..., \mu_k \leftarrow$ centroïdes initiaux pris aléatoirement dans X ;
tant que *non convergence* **faire**
  pour $i = 1...k$ **faire**
    $C_i \leftarrow \{x \in X : i = \text{argmin}_j \|x - \mu_j\|\}$;
  pour $i = 1...k$ **faire**
    $\mu_i \leftarrow \frac{1}{|C_i|} \sum_{x \in C_i} x$;

On peut alors montrer le résultat suivant.

Lemme 17.1 *La fonction objectif des k-moyennes décroît strictement à chaque itération de l'algorithme ci-dessus.*

Démonstration. On pose $G(C_1, ..., C_k)$ la fonction objectif, que l'on peut réécrire :

$$G(C_1, ..., C_k) = \min_{\mu_1,...,\mu_n \in \mathcal{X}} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2 \tag{17.1}$$

On pose $\mu(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x$. On remarque :

$$\mu(C_i) = \text{argmin}_{\mu \in \mathcal{X}} \sum_{x \in C_i} \|x - \mu\|^2 \tag{17.2}$$

(Pour le montrer, il suffit de remarquer que la fonction est strictement convexe, et on regarde quand le gradient s'annule).

On peut alors réécrire :

$$G(C_1, ..., C_k) = \sum_{i=1}^{k} \sum_{c \in C_i} \|x - \mu(C_i)\|^2$$

On considère la mise à jour à l'itération t. On note $C_1^{(t-1)}$,...,$C_k^{(t-1)}$ la partition précédente, et soit $\mu_i^{(t-1)} = \mu_i(C_i^{(t-1)})$, et soit $C_1^{(t)}$,...,$C_k^{(t)}$ la nouvelle partition à l'itération t. En utilisant l'équation 17.1

$$G(C_1^{(t)}, ..., C_k^{(t)}) \leq \sum_{i=1}^{k} \sum_{x \in C_i^{(t)}} \|x - \mu_i^{(t-1)}\|^2 \tag{17.3}$$

Or, la nouvelle partition $(C_1^{(t)}, ..., C_k^{(t)})$ minimise l'expression $\sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i^{(t-1)}\|^2$ parmi tous les partitionnements possibles. Donc

$$\sum_{i=1}^{k} \sum_{x \in C_i^{(t)}} \|x - \mu_i^{(t-1)}\|^2 \leq \sum_{i=1}^{k} \sum_{x \in C_i^{(t-1)}} \|x - \mu_i^{(t-1)}\|^2$$

On utilise l'équation 17.2 pour le membre droite, et l'équation 17.3 puis 17.2 sur le membre gauche, on a :

$$G(C_1^{(t)}, ..., C_k^{(t)}) \leq G(C_1^{(t-1)}, ..., C_k^{(t-1)})$$

□

Développement 18

Un algorithme d'approximation pour un problème de clustering

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Ce développement propose un algorithme d'approximation pour un problème de clustering consistant à partitionner un ensemble de points en minimisant le diamètre de chaque cluster. Ainsi, ce développement s'insère dans les leçons 11 et 14.

Introduction. On considère le problème de clustering suivant :

Définition 18.1 (Point Clustering (PC))
Données *: ensemble S de n points dans un espace métrique $(\mathcal{X}, d)$.*
Sortie *: partition de S en k ensembles $C_1, ..., C_k$ qui minimise*

$$\max_{1 \leq i \leq k} \delta(C_i)$$

où $\delta(C) = \max_{x,x' \in C} d(x, x')$ est le diamètre d'un ensemble de points C.

Un premier algorithme. On suppose le diamètre optimal Δ^* connu. On considère l'algorithme suivant :

Algorithme 18.1 : PartitionAux(S,k, Δ^*)

$i \leftarrow 1$;
tant que $i \leq k$ *et* $S \neq \emptyset$ **faire**
  $p_i \leftarrow \text{Random}(S)$;
  $C_i \leftarrow \{p' | d(p_i, p') \leq \Delta^*\}$;
  $S \leftarrow S \backslash C_i$;
  $i \leftarrow i + 1$;
retourner $C_1, ..., C_k$

On veut montrer que l'algorithme ci-dessus est une 2-approximation. Pour cela, il faut tout d'abord montrer qu'il retourne bien une partition de S.

Lemme 18.1 *L'algorithme 18.1 renvoie une partition de l'ensemble S en entrée.*

Démonstration. Par l'absurde, on suppose qu'il existe un point q qui n'est pas retenu par l'algorithme. On remarque alors que tous les C_i sont non vides (et donc les p_i sont bien définis). Ainsi, pour tout $1 \leq i \leq k$,

on a $d(p_i, q) > \Delta^*$, et de plus, pour tout $1 \leq i < j \leq k$, on a $d(p_i, p_j) > \Delta^*$. Ainsi, $\{p_1, ..., p_k, q\}$ est un sous-ensemble de S contenant $k+1$ points tous distants d'au moins strictement Δ^*. Un tel ensemble ne peut pas être partitionner en k clusters de diamètre au plus Δ^*, donc S non plus. Or, S peut être partitionné de la sorte par hypothèse, c'est absurde. □

Proposition 18.1 *L'algorithme 18.1 est une 2-approximation du problème* **PC**.

Démonstration. Par le lemme précédent, l'algorithme retourne bien une partition $C_1, ..., C_k$ de l'ensemble S en entrée. Soit $1 \leq i \leq k$ et $x, x' \in C_i$, on a :

$$d(x, x') \leq d(x, p_i) + d(p_i, x') \leq 2\Delta^*$$

Ainsi, l'algorithme est bien une 2-approximation. □

Cas général. On ne suppose plus connaître Δ^*. On considère alors l'algorithme qui sélectionne les centres successivement en prenant le point le plus loin des précédents, puis qui assigne chaque point au centre le plus proche.

Algorithme 18.2 : Partition(S,k)

```
q_1 ← Random(S) ;
pour i = 2...k faire
    q_i ← argmax_{x∈S} d(x, {q_1, ..., q_{i-1}}) ;
pour x ∈ S faire
    i ← argmin_{1≤i≤k} d(x, q_i) ;
    C'_i ← C'_i ∪ {x} ;
retourner C'_1, ..., C'_k
```

Remarque 18.1 *Dans l'algorithme, on considère une notion de distance à un ensemble $d(x, E)$ où $x \in S$ et $E \subset S$. Plusieurs définitions sont possibles, mais on utilisera ici*

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

Proposition 18.2 *L'algorithme 18.2 est une 2-approximation du problème* **PC**.

Démonstration. On compare les exécutions des algorithmes 18.1 et 18.2. Si, lors de l'exécution du premier, les centres sélectionnés par le second sont disponibles, alors la proposition précédente est correcte.

On suppose que ce n'est pas le cas, c'est-à-dire qu'il existe $1 < i \leq k$ avec $q_i \in \bigcup_{j=1}^{i-1} C_i$ et $p_j = q_j$ pour $1 \leq j < i$. Ainsi, il existe un indice $1 < l < i$ avec $q_i \in C_l$, et donc $d(q_i, p_l) \leq \Delta^*$. Or, l'algorithme 18.2 choisit le point le plus loin pour nouveau centre. Autrement dit,

$$q_i = \text{argmax}_{x \in S} d(x, \{q_1, ..., q_{i-1}\})$$

Ainsi, pour tout $x \in S$, on a $d(x, \{q_1, ..., q_{i-1}) \leq d(q_i, \{q_1, ..., q_{i-1}) \leq d(q_i, q_l) \leq \Delta^*$. Donc tous les points de S sont à une distance au plus Δ^* des centres déjà sélectionnés. Avec un raisonnement similaire que pour l'algorithme précédent, on a bien une 2-approximation dans ce cas. □

Développement 19

Résolution d'un exercice avancé de SQL

Auteur·e·s: Emile Sorci, Marin Malory
Références : [Silberschatz et al., 2020]

Ce développement est la présentation et la correction d'un exercice avancé de SQL. Il permet notamment de présenter la clause `WITH ... AS ...` *qui simplifie certaines requêtes. Il s'intègre parfaitement dans les leçons 23 et 18.*

Exercice 19.1 *On considère une base de donnée sur des entreprises, ainsi qu'une série de requêtes sur cette base.*

La base de données contient les tables suivantes :

— `Employes(id, nom, rue, ville)`

— `Travail(id, nom_compagnie, salaire)`

— `Compagnie(nom_compagnie, ville)`

Donner une expression SQL pour chacune des requêtes suivantes :

1. *Trouver l'identifiant, le nom de la ville de résidence de chaque employé travaillant pour « Banque Centrale ».*
2. *Trouver l'identifiant, le nom et la ville de résidence de chaque employé travaillant pour « Banque Centrale » et gagnant plus de 10 000 €.*
3. *Trouver l'identifiant de chaque employé qui gagne plus que n'importe quel employé de la « Banque Locale ».*
4. *On suppose que des compagnies peuvent se situer dans plusieurs villes. Trouver le nom de chacune des compagnies qui ne sont situées que dans des villes où la « Banque Locale » est implantée.*

Pour les questions suivantes, on utilisera la clause `WITH`*, qui s'utilise de la manière suivante* `WITH nom_table(att_1,...,att_n) AS (sous-requête) requête` *et qui permet de renommer la table obtenue via la sous-requête.*

5. *Trouver le nom de la (ou les) compagnie(s) qui a(ont) le plus d'employés.*
6. *Trouver le nom de chaque compagnie où, en moyenne, un employé gagne plus qu'un salarié de « Banque Centrale ».*

Solution.

1.
```
SELECT e.id, e.ville
FROM Employes as e
JOIN Travail as t on e.id=t.id
WHERE t.nom_compagnie = "Banque Centrale"
```

2.
```
SELECT e.id, e.ville
FROM Employes as e
JOIN Travail as t on e.id=t.id
WHERE t.nom_compagnie = "Banque Centrale" AND t.salaire >= 10000
```

3.
```
SELECT e.id
FROM Employes AS e
JOIN Travail AS t on e.id=t.id
WHERE t.salaire >= (
        SELECT MAX(salaire)
        FROM Travail
        WHERE nom_compagnie = "Banque Locale"
)
```

4.
```
SELECT nom_compagnie
FROM Compagnie

EXCEPT

SELECT c.nom_compagnie
FROM Compagnie AS c
LEFT JOIN (
    SELECT DISTINCT ville
    FROM Compagnie
    WHERE nom_compagnie = "Banque locale"
     ) AS v
ON c.ville = v.ville
WHERE v.ville IS NULL;
```

5.
```
WITH NbEmployes(nom_compagnie,n) AS (
  SELECT nom_compagnie, COUNT(id) AS n
  FROM Travail
  GROUP BY nom_compagnie
)

SELECT t.nom_compagnie
FROM NbEmployes AS t
WHERE t.n >= (
  SELECT MAX(n)
  FROM NbEmployes
)
```

6.
```
WITH travail_salaire(nom_compagnie, salaire_moyen) AS (
  SELECT nom_compagnie, AVG(salaire)
  FROM Travail
  GROUP BY nom_compagnie
)

SELECT t.nom_compagnie
FROM travail_salaire AS t
WHERE t.salaire_moyen > (
  SELECT salaire_moyen
```

```
  FROM travail_salaire
  WHERE nom_compagnie = "Kinder"
)
```

Développement 20

Un algorithme d'approximation glouton pour un problème d'ordonnancement

Auteur·e·s: Bertrand Jules
Références : [Benoit et al., 2013]

Ce développement propose un algorithme d'approximation pour un problème d'ordonnancement de tâches indépendantes. Ainsi, il s'intègre dans les leçons 11 et 13. Cet algorithme étant glouton, il illustre aussi cette stratégie pour la leçon 12.

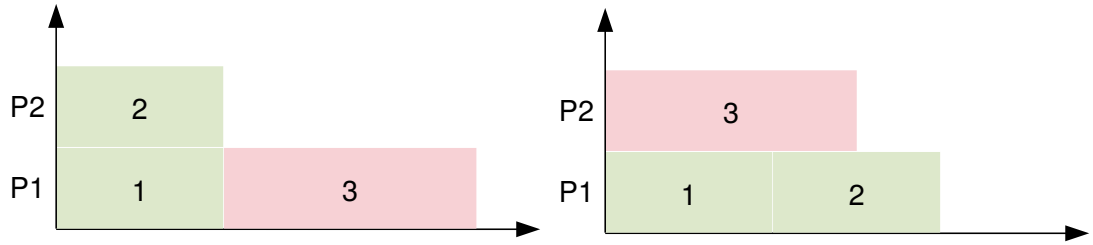
Problème. On considère le problème d'ordonnancement suivant.

Définition 20.1 (INDEP(p))
Données : *n tâches* $\{T_1, ..., T_n\}$ *et* $p \geq 2$ *processeurs. À chaque tâche* T_i *est associée un poids* $w(T_i)$.
Sortie : *ordonnancement des tâches* $\mathbf{alloc} : \{T_1, ..., T_n\} \to \{1, ..., p\}$, *tel que*

$$\max_{1 \leq l \leq p} \sum_{T_i \in \mathbf{alloc}^{-1}(l)} w(T_i)$$

soit minimal.

Exemple 20.1 *On prend 4 tâches, avec* $w(T_1) = 2$, $w(T_2) = 2$, $w(T_3) = 3$.



On considère l'algorithme **glouton-online** qui, pour toute tâche $T_1, ..., T_n$, l'affecte au processeur le plus libre. Sur l'exemple précédent, l'ordonnancement obtenu via **glouton-online** correspond au schéma de gauche. On remarque directement que cet algorithme n'est pas optimal.

Théorème 20.1 (Résultat d'approximation)
glouton-online *est une* $\left(2-\frac{1}{p}\right)$*-approximation pour le problème* $\mathbf{INDEP}(p)$*. Le facteur d'approximation est atteint.*

Notations. On note τ^* le temps d'exécution optimal d'un ordonnancement de ces tâches et $S = \sum_{i=1}^{n} w(T_i)$.

Lemme 20.1 $\tau^* \geq \frac{1}{p}\sum_{i=1}^{n} w(T_i)$

Démonstration. On note $\tau_1, ..., \tau_p$ les temps d'exécutions sur chaque processeur, et on a alors $S := \sum_{i=1}^{n} w(T_i) = \sum_{k=1}^{p} \tau_k$.

On sait que $\tau^* = \max_{1\leq k\leq p} \tau_k$. Ainsi,

$$S \leq \sum_{k=1}^{p} \tau^*$$

La lemme en découle directement. □

Preuve du théorème. On suppose que l'algorithme **glouton-online** retourne un ordonnancement $\mathbf{alloc}$ de nos n tâches, de temps d'exécution total τ. On veut montrer que

$$\frac{\tau}{\tau^*} \leq 2 - \frac{1}{p}$$

On suppose sans perdre de généralité que $\tau = \tau_1$, c'est-à-dire que c'est le premier processeur qui termine en dernier. On note T_j la dernière tâche exécutée sur P_1, et on note $\tau_0 = \tau_1 - w(T_j)$, c'est-à-dire le temps d'exécution avant qu'on affecte T_j à P_1.

On remarque alors que pour tout $2 \leq i \leq p$, $\tau_0 \leq \tau_i$. En, effet, s'il existe $2 \leq i \leq p$ tel que $\tau_0 > \tau_i$, alors la tâche T_j n'aurait pas été affecté à P_1.

Il sera judicieux d'illustrer avec un schéma sur lequel apparaît T_j*,* τ_0 *et* τ_1*.*

$$\begin{aligned} S &= \sum_{i=1}^{p} \tau_i \\ &= \tau_1 + \sum_{i=2}^{p} \tau_i \\ &\geq \tau_1 + (p-1)\tau_0 \\ &= p\tau_1 - (p-1)w(T_j) \\ &= p\tau - (p-1)w(T_j) \end{aligned}$$

Or, on remarque que $w(T_j) \leq \tau^*$, et par le lemme précédent, $\tau^* \geq \frac{1}{p}S$. Ainsi,

$$\tau \leq \frac{S}{p} + \frac{p-1}{p} w(T_j) \leq \tau^* + \frac{p-1}{p}\tau^* = \left(2 - \frac{1}{p}\right)\tau^*$$

Montrons que le facteur est atteint. Pour cela on considère $p(p-1)$ tâches de taille 1 et une tâche de taille p. La solution optimale consiste à affecter la tache de taille p à un processeur, et p tâches de taille 1 aux $p-1$ processeurs restants. On obtient $\tau^* = p$.

La solution gloutonne qui prend la grande tâche en dernier va alors remplir les p processeurs de manière équilibrée avec les $p(p-1)$ premières tâches. Ainsi, lorsqu'on affecte la dernière tâche, tous les processeurs sont remplis avec $p-1$ tâches et ainsi $\tau = p-1+p = 2p-1$. Finalement,

$$\frac{\tau}{\tau^*} = \frac{2p-1}{p} = 2 - \frac{1}{p}$$

On pourra mentionner le fait que l'algorithme glouton offline, qui trie les tâches avant de les exécuter n'est pas optimal non plus, mais donne un meilleur facteur d'approximation.

Développement 21

Algorithmes onlines de remplacement de pages

Auteur·e·s: Marin Malory
Références : [Motwani and Raghavan, 1995]

Ce développement étudie deux algorithmes de remplacement de pages : LRU et FIFO. En particulier, on s'intéresse au moyen de mesurer la performance de tels algorithmes onlines. Le problème original justifie ce développement pour les leçons 15 et 16. De plus, si la présentation est plus orientée sur FIFO, il s'insère dans la leçon 5. Ces deux algorithmes utilisant la stratégie gloutonne, ce développement illustre la leçon 12. Enfin, le problème de pagination peut être vu comme un problème de gestion de ressources, et s'insère ainsi dans la leçon 13.

Problème de pagination online. On considère une mémoire à deux niveaux : un **cache** (ou mémoire rapide) de taille k, et une **mémoire principale** (ou mémoire lente) pouvant potentiellement garder une infinité d'éléments en mémoire (on peut aussi fixer sa taille très grande devant k).

Un **algorithme de pagination** décide quel élément garder en mémoire cache à tout instant. On a une séquence de requêtes $\rho = (\rho_1, ..., \rho_N)$, chacun spécifiant un élément mémoire ;

— si l'élément ρ_i est dans le cache, on a un *cache hit* ;

— si l'élément ρ_i n'est pas dans le cache, on a un *cache miss*.

On veut créer un algorithme online qui minimise le nombre de cache miss.

Remarque 21.1 *Un algorithme online n'a pas accès aux requêtes futures, c'est la différence avec un algorithme offline qui connaît à l'avance toutes les requêtes.*

On considère trois algorithmes déterministes classiques utilisés en architecture et systèmes d'exploitations :

— LRU (Least Recently Used) : on enlève du cache l'élément dont l'utilisation la plus récente est arrivée le plus tôt

— FIFO (First In First Out) : on enlève le premier arrivé dans le cache.

— LFU (Least Frequently Used) : on enlève celui utilisé le moins souvent

On va se concentrer sur les deux premiers.

Un peu de vocabulaire. Étant donné un algorithme de pagination A, et une séquence de requêtes $\rho = (\rho_1, ..., \rho_N)$, on note $f_A(\rho)$ le nombre de cache miss de A sur la séquence ρ.

On note de plus $f_O(\rho)$ le nombre de cache miss de l'algorithme offline optimal MIN.

Remarque 21.2 *L'algorithme MIN a connaissance de la totalité de ρ, il supprime du cache l'élément qui apparaît le plus longtemps après dans ρ. La preuve d'optimalité est non trivial.*

Remarque 21.3 *Sans perdre de généralité, on considère que le cache est toujours vide au départ (les k premières requêtes entraînent toujours un cache miss).*

Comment mesurer la performance d'un tel algorithme. Une première idée consiste à faire comme pour la complexité : étudier le pire cas. Étant donné un algorithme A, on définie la performance dans le pire cas $PC(A)$ comme :

$$PC(A) = \max_{\rho \in R_N} f_A(\rho)$$

Cette mesure n'est pas très utile, comme le montre le théorème suivant.

Théorème 21.1 *On considère qu'il n'y a que $k+1$ éléments mémoires et on considère des séquences de requêtes de taille N. Pour tout algorithme déterministe de pagination A,on a $PC(A) = N$.*

Démonstration. Soit $\rho = (\rho_1, ..., \rho_k)$ une séquence de départ avec les ρ_i distincts deux à deux. La mémoire cache est alors remplie. On demande le $k+1$-ème élément mémoire ρ_{k+1}, et A supprime un élément que l'on nomme ρ_{k+2}. On demande ensuite ρ_{k+2} qui n'est plus dans le cache, et on recommence jusqu'à ρ_N. La séquence se construit alors par induction sur N. □

Compétitivité. On va chercher un autre moyen de mesurer la performance des algorithmes, en le comparant à l'optimal offline MIN.

Définition 21.1 (Compétitivité) *Un algorithme de pagination déterministe A est C-compétitif s'il existe b indépendant de N tel que pour toute séquence de requêtes ρ,*

$$f_A(\rho) - C f_0(\rho) \leq b$$

On note C_A la borne inférieure des C tel que A est C-compétitif.

On va alors pouvoir distinguer nos algorithmes.

Théorème 21.2 *LRU et FIFO sont k-compétitifs.*

Théorème 21.3 *LFU n'a pas de coefficient de compétitivité borné.*

Théorème 21.4 (Optimalité) *Pour tout algorithme déterministe de pagination A, $C_A \geq k$.*

Démonstration du théorème 21.2. Soit $\rho = (\rho_1, ..., \rho_N)$. On partitionne ρ en phases de la manière suivante : P_0 est la phase vide, et pour $i > 0$, P_i est la séquence maximale suivant P_{i-1} contenant au plus k requêtes distinctes.

Exemple 21.1 *Pour* $k = 2$ *et* $\rho = (1, 1, 2, 3, 3, 4, 5)$, *on a* $P_0 = \emptyset$, $P_1 = (1, 1, 2)$, $P_2 = (3, 3, 4)$ *et* $P_3 = (5)$.

On numérote les phases pour $0 \leq i \leq m$, et on montre que pour tout i, $f_A(P_i) \leq k$ pour $A \in \{LRU, FIFO\}$.

Par l'absurde, si $f_A(P_i) > k$, alors il existe un élément mémoire α qui provoque un cache miss par deux fois dans la phase P_i. On montre que cela est impossible via les algorithmes LRU et FIFO. Entre les deux moments où α provoque un cache miss, les requêtes concernent au plus les $k-1$ autres éléments mémoires de la phase P_i.

— Cas LRU : au moment de l'éviction de α (entre les deux caches miss), α avait la dernière utilisation la plus tôt. Ainsi, les $k-1$ autres éléments du caches ont eu une requête entre le premier cache miss de α et son éviction. Puisque lors de son éviction, la requête concernait un élément non présent en cache, on obtient $k+1$ éléments mémoires distincts demandés durant P_i.

— Cas FIFO : même raisonnement. En effet, lors de l'éviction de α, les $k-1$ autres éléments sont en haut de la file, α en bas et on insère un $k+1$-ème élément dans la file, ce qui est absurde.

Ainsi, on a

$$f_A(\rho) = \sum_{i=1}^{m} f_A(P_i) \leq mk$$

Maintenant, montrons que $f_o(\rho) \geq m - 1$. Soit $0 < i < m$, on pose q la première requête de P_i, et q' la première requête de P_{i+1}. On considère la séquence qui démarre juste après q et termine juste après q'. On montre que MIN fait au moins une erreur sur cette séquence. En effet, q est chargé en mémoire cache au début, et cette séquence contient au moins k éléments distincts (q' et P_i sans q). Ainsi, au moins un de ces éléments ne sera pas dans le cache. Ainsi, on obtient au moins $(m-1)$ cache miss.

Finalement,

$$f_A(\rho) - kf_o(\rho) \leq mk - k(m-1) = k$$

Ce qui conclut la preuve. □

Pour rendre la preuve plus claire, il sera judicieux de faire un schéma temporel des différentes requêtes. De plus, il s'avère que la preuve est beaucoup plus claire pour FIFO que LRU.

Exercice 21.1 (Démonstration du théorème 21.3)
Montrer le théorème 21.3 en posant pour $l \geq 0$ *:*

$$\rho = \rho_1^l \rho_2^l ... \rho_{k-1}^l (\rho_k \rho_{k+1})^{l-1}$$

Pratique vs Théorie. En pratique, on sait bien que LRU marche mieux que FIFO, alors que théoriquement leurs performances sont similaires. Ce modèle de performance est donc critiquable. Pour distinguer théoriquement FIFO et LRU, il faut alors notamment limiter la capacité de l'adversaire, et prendre en compte les principes de localités.

Développement 22

Construction d'un additionneur à retenue anticipée

Auteur·e·s: Sorci Émile, Marin Malory
Références : [Hennessy and Patterson, 2012]

Ce développement présente la construction d'un additionneur rapide : l'additionneur à retenue anticipée (carry-lookahead adder). Ce développement est parfait pour présenter un circuit combinatoire complexe, illustrant la leçon 19. De plus, ce circuit utilise un paradigme diviser-pour-régner, illustrant ainsi la leçon 12 en présentant une implémentation matérielle d'un algorithme. Enfin, ce circuit illustre la manière dont les opérations arithmétiques sont réalisées en machine, et illustre ainsi certains principes de fonctionnement des ordinateurs, illustrant la leçon 20.

Introduction. Un additionneur n bits est un circuit combinatoire : il peut être écrit sous le forme d'une formule logique. L'additionneur classique (*full-adder*) présente un problème majeur : pour calculer le i-ème bit du résultat, on doit attendre la retenue sortante c_i. Dans ce circuit, le chemin critique est alors de taille $\mathcal{O}(n)$.

On pourra ici dessiner un full-adder et surligner le chemin critique de longueur n.

$$s_i = a_i\overline{b_i}\,\overline{c_i} + \overline{a_i}b_i\overline{c_i} + \overline{a_i}\,\overline{b_i}c_i + a_ib_ic_i \tag{22.1}$$

$$c_{i+1} = a_ib_i + a_ic_i + b_ic_i \tag{22.2}$$

On peut alors réécrire l'équation précédente :

$$c_{i+1} = g_i + p_ic_i,\ g_i = a_ib_i,\ p_i = a_i + b_i \tag{22.3}$$

— si g_i est vraie, alors c_{i+1} est vraie, et donc une retenue est **générée** ;

— si p_i est vraie, alors si c_i est vraie, alors on a c_{i+1} qui est vraie, la retenue a été **propagée**.

En itérant, on a alors :

$$c_{i+1} = g_i + p_ig_{i-1} + p_ip_{i-1} + p_ip_{i-1}g_{i-2} + ... + p_ip_{i-1}...p_1p_0g_0 \tag{22.4}$$

CLA. L'additionneur qui calcule les retenues en utilisant l'équation 22.4 s'appelle **additionneur à retenue anticipée**, ou CLA pour *carry-Lookahead adder*. Malheureusement, cette formule n'est pas utilisable directement telle quelle (il fait faire un `OR` à n entrée et un `AND` à n entrées, et des fils très longs).

On va donc travailler par bloc :

— P_{ij} : une retenue est propagée de i à j.

— G_{ij} : une retenue est générée entre i et j ;

On peut les définir par récurrence de la manière suivante :

$$P_{i,j+1} = P_{i,j}P_{j+1} \tag{22.5}$$

$$P_{i,j+1} = g_{j+1} + p_{j+1}G_{i,j} \tag{22.6}$$

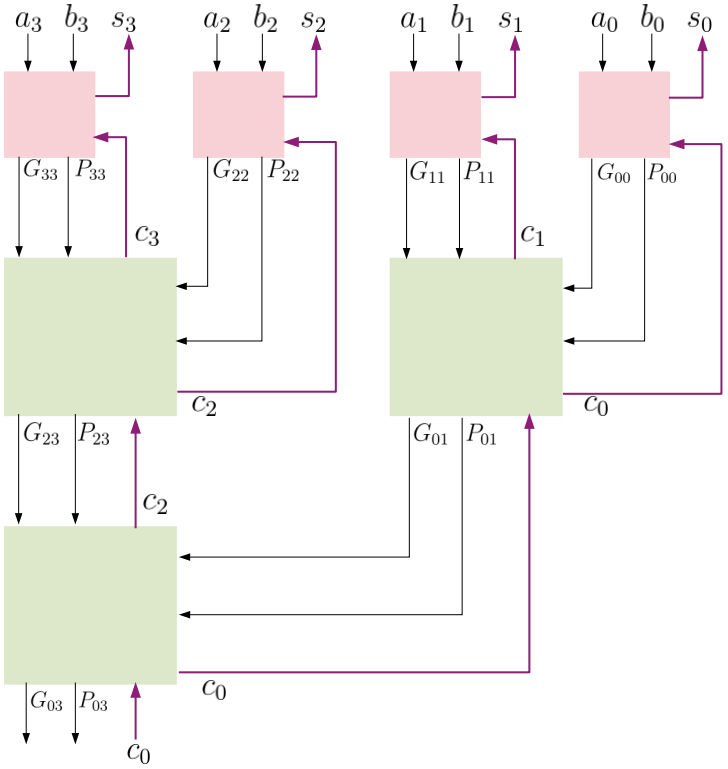
On a alors pour tout $i \leq j \leq k-1$:

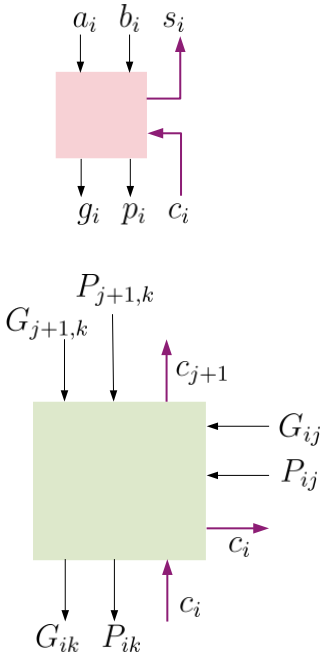
$$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij} \tag{22.7}$$

$$P_{ik} = P_{ij}P_{j+1,k} \tag{22.8}$$

$$c_{j+1} = G_{ij} + P_{ij}c_i \tag{22.9}$$

On peut alors dessiner un additionneur à retenue anticipée sur 4 bits.





En notant c_v et c_r les longueurs des chemins critiques dans dans les blocs vert et rouge, on remarque que pour n bits, la longueur du chemin critique $l_{CLA}(n)$ vérifie l'inéquation :

$$l_{CLA}(n) = c_v + l_{CLH}(n/2)$$

et $l_{CLA}(1) = c_r$. Ainsi, on a, on supposant que n est une puissance de 2 :

$$l_{CLA} = \log_2(n)c_v + c_r = \mathcal{O}(\log_2(n))$$

Développement 23

Recherche de chemin critique dans un circuit booléen

Auteur·e·s: Marin Malory
Références : [Cormen et al., 2009]

Ce développement propose un algorithme permettant de calculer la profondeur d'un graphe orienté acyclique, et montre sa correction. Cet algorithme trouve son application dans le calcul du délai d'un circuit booléen, celui-ci étant proportionnel à la longueur du chemin critique du circuit. Ainsi, ce développement s'insère dans les leçons 7 et 19.

Introduction. La recherche d'un chemin critique dans un circuit booléen revient au calcul de la profondeur d'un DAG (graphe orienté acyclique). Un tel algorithme peut trouver application dans le cadre d'un logiciel de dessin de circuit booléen, permettant ainsi de connaître la longueur du chemin critique et d'identifier de tels chemins.

L'idée de l'algorithme proposé ici est de simplifier et modifier un algorithme classique : l'algorithme des plus courts chemins dans un DAG pondéré.

Algorithme. On présente ici un algorithme linéaire qui calcule la profondeur d'un DAG, ainsi que sa correction totale. Il repose notamment sur un tri topologique du graphe.

Algorithme 23.1 : CheminCritique(G)

Données : graphe orienté acyclique $G = (V, E)$, où $n = |V|$.
Résultat : profondeur de G
$(v_i)_{1 \leq i \leq n} \leftarrow$ Tri-Topologique(G) ;
pour $i = 1...n$ **faire**
  dist$[v_i] \leftarrow 0$;
pour $i = 1...n$ **faire**
  dist$[v_i] \leftarrow \max_{uv_i \in E}(\text{dist}[u] + 1)$;
retourner $\max_{1 \leq i \leq n}$ *dist*$[v_i]$;

Proposition 23.1 *L'algorithme 23.1 s'exécute en temps linéaire en la taille du graphe.*

Démonstration. Étant donné un DAG $G = (V, E)$, son tri topologique se réalise en temps $\mathcal{O}(|V| + |E|)$, donc linéaire en la taille du graphe. La première boucle s'exécute en $|V|$ opérations élémentaires. Ensuite, en notant pour $d(u)$ le degré entrant d'un sommet $u \in V$, l'exécution de la seconde boucle réalise

$$\sum_{i=1}^{n} (d(v_i) - 1) + 1 = |E|$$

opérations élémentaires, en comptant à chaque itération le calcul du maximum et l'affectation. Enfin, on calcule le résultat final en $|V| - 1$ comparaisons.

Finalement, l'algorithme s'exécute en temps linéaire. □

La terminaison de l'algorithme est directe. On montre ici sa correction.

Proposition 23.2 *L'algorithme 23.1 est correct.*

Démonstration. On montre l'invariant de boucle suivant (pour la seconde boucle) : « À la fin de chaque itération, dist$[v_j]$ $(1 \leq j \leq i)$ contient la longueur du plus long chemin terminant en v_j.

Initialisation À la première itération, on a dist$[v_1] = \max \emptyset = 0$. En effet, puisque G est un DAG, v_1 est une source et n'a pas d'antécédent dans le graphe.

Conservation On suppose l'invariant vrai à l'itération $i-1$, montrons qu'il l'est à l'itération i. Pour $1 \leq j \leq i-1$, par hypothèse, dist$[v_j]$ est déjà la longueur du plus chemin terminant en v_j. Montrons qu'un plus long chemin terminant en v_i est de longueur dist$[v_i] = \max_{uv_i \in E}(\text{dist}[u] + 1)$.

Soit μ un plus long chemin terminant en v_i. Par tri topologique de G, on peut écrire $\mu = v_{i_1}...v_{i_k}$ avec $k \leq i$ la longueur du chemin et $v_{i_k} = v_i$. On remarque alors que $\mu' = v_{i_1}...v_{i_{k-1}}$ est un plus long chemin terminant en $v_{i_{k-1}}$ par maximalité de μ. Par hypothèse, on a alors $k - 1 = \text{dist}[v_{i_{k-1}}]$. Ainsi, $\max_{uv_i \in E}(\text{dist}[u] + 1) \geq \text{dist}[v_{i_{k-1}}] + 1 = k$. Par l'absurde, si dist$[v_i] > k$, il existe un arc $uv_i \in E$ tel que dist$[u] \geq k$. Par l'invariant, il existe un plus long chemin terminant en u de longueur $\geq k$, et on peut donc construire un chemin terminant en v_i de longueur $> k$, ce qui contredit la maximalité de μ.

Ainsi, en sortie de boucle, dist$[u]$ contient la longueur des plus longs chemins terminant en $u \in V$. Finalement, $\max_{u \in V}$ dist$[u]$ est bien la profondeur du DAG. □

Conclusion. Cet algorithme nous permet de calculer la profondeur du graphe. À partir du tableau obtenu, on peut aussi retourner l'ensemble des chemins critiques. Pour cela, il suffit pour chaque sommet qui maximise dist$[u]$, de remonter dans le DAG par les ancêtres v de u tels que dist$[v] =$ dist$[u] - 1$ et ainsi de suite. Dans le cadre d'un logiciel, cette modification permettrait à l'utilisateur de colorier les chemins critiques sur son circuit.

Développement 24

Validation croisée

Auteur·e·s: Marin Malory
Références : [Shalev-Shwartz and Ben-David, 2014]

Ce développement présente la méthode de la validation croisée, ainsi qu'un exemple où celle-ci échoue. Cette méthode est particulièrement utilisée en apprentissage machine, illustrant ainsi la leçon 24. De plus, cette méthode permet de tester un modèle sur des données de manière astucieuse, et illustre la leçon 3 si ce genre de tests sont abordés.

Objectif. Pour valider un modèle, la méthode classique consiste à réserver une partie des données seulement pour cette partie. Cependant, lorsque les données sont rares, on ne veut pas en gaspiller. La méthode de validation croisée (k-fold) permet d'estimer la performance de notre modèle sans perdre trop de données.

Principe. Dans une validation croisée k-fold, on sépare notre jeu d'entraînement (les 80% restant après le retrait des données servant au test) en k sous-ensembles de taille n/k. Pour chaque sous-ensemble, on entraîne notre algorithme sur les données restantes et vérifie sur celui-ci. Enfin, on estime l'erreur en prenant la moyenne de toutes les erreurs.

Remarque 24.1 *On rappelle que l'erreur est une mesure définie avec le problème. Ainsi, étant donné un ensemble de test $S \in (\mathcal{X} \times \mathcal{Y})^n$, et une fonction $h : \mathcal{X} \to \mathcal{Y}$, on notera l'erreur $L_S(h)$. Si $\mathcal{Y}$ est discret :*

$$L_S(h) = \frac{1}{n}|\{1 \leq i \leq n | h(x_i) \neq y_i\}|$$

et si $\mathcal{Y}$ est continue, on peut prendre :

$$L_S(h) = \frac{1}{n}\sum_{i=1}^{n}(h(x_i) - y_i)^2$$

Remarque 24.2 *Lorsque $k = n$, on parle de méthode leave-one-out (LOO).*

Sélection de modèle. La validation croisée est principalement utilisée pour de la validation de modèle. On a donc un algorithme A, et un ensemble de paramètres Θ. L'algorithme de sélection est alors l'algorithme 46 ci-dessous.

Complexité. La complexité est :

$$C_{VC-A}(n, k, \Theta) = k\sum_{\theta \in \Theta} C_A(n, \theta)$$

Algorithme 24.1 : Validation croisée k-folds - sélection de modèle.

Données : ensemble d'entraînement $S = (x_1, y_1), ..., (x_n, y_n)$
ensemble de paramètres Θ
algorithme d'apprentissage A
entier k
$S_1, ..., S_k \leftarrow$ Partition(S) ;
pour $\theta \in \theta$ **faire**
  pour $i = 1...k$ **faire**
    $h_{i,\theta} = A(S \backslash S_i, \theta)$;
    erreur$(\theta) \leftarrow \frac{1}{k} \sum_{i=1}^{k} L_{S_i}(h_{i,\theta})$;
$\theta^* \leftarrow$ argmin$_\theta$[erreur(θ)] ;
retourner $A(S; \theta^*$

Exemple. Si on considère l'algorithme des k voisins, on peut prendre $\Theta = \{1, ..., n\}$ où n est le nombre de points. Ce choix n'est pas bon en pratique à cause de la complexité.

Considérons une implémentation des k voisins naïve : on calcule toutes les distances en n opérations élémentaires, et on cherche les K minimums avec un tableau auxiliaire de taille k. On récupère alors les k minimums en parcourant les $n - k$ sommets restants et en retirant le maximum du tableau. On obtient une complexité au plus $(n - k)k$.

Ainsi, en notant $C_{\text{VC-kNN}}$ la complexité de la validation croisée appliquée à l'algorithme des k voisins, on a :

$$C_{\text{VC-kNN}} \leq k \sum_{i=1}^{n} i(n - i) + n = \mathcal{O}(kn^3)$$

Il faut donc faire un choix raisonnable de Θ, puisqu'en général prendre trop de voisins mène à un sous-apprentissage.

Efficacité. La validation croisée fonctionne bien en pratique. Mais cela peut échouer, comme le montre le cas suivant, qui est exemple un peu artificiel.

On suppose qu'on dispose d'un jeu de données $S \in (\mathcal{X} \times \mathcal{Y})^n$. Pour tout $y \in \mathcal{Y}$, l'étiquette est choisi uniformément et aléatoirement parmi $\{0, 1\}$. On considère un algorithme d'apprentissage A qui retourne le classifieur constant $h : x \mapsto 1$ si la parité des étiquettes sur l'ensemble d'entraînement vaut 1, et le classifieur nul sinon.

La différence entre l'erreur estimée par la validation croisée n-fold (FOO) et la vraie erreur est toujours $1/2$.

En effet, la vraie erreur est $\mathbb{E}_S(L_S(h)) = n/2$ car h est constante égale à 1 ou 0 avec probabilité $1/2$.

Soit $S = (x_1, y_1), ..., (x_n, y_n)$ un jeu de test et $h = A(S)$. On note $S_i = S \backslash \{(x_i, y_i)\}$ et $h_i = A(S_i)$. On note n_1 (resp. n_0) le nombre de 1 dans S, et sans perdre de généralité, on suppose n_1 impair.

Soit $1 \leq i \leq n$.

— Si $y_i = 1$, alors S_i contient un nombre pair de 1, et donc $h_i = 0$. On a alors $L_{S_i}(h_i) = 1$.

— Si $y_i = 0$, alors S_i contient un nombre impair de 1, et donc $h_i = 1$, et $L_{S_i}(h_i) = 1$.

On raisonne de même si n_1 est pair. Finalement,

$$\mathbb{E}_S \left(\frac{1}{n} \sum_{i=1}^{n} L_{S_i}(h_i) \right) = 1$$

et donc la différence d'erreur est toujours $1/2$.

Développement 25

Vérification du produit de matrice

Auteur·e·s: Marin Malory
Références : [Mitzenmacher and Upfal, 2005]

Ce développement présente un exemple d'utilisation d'aléatoire afin de vérifier si un produit de matrice est correct de manière efficace. Il s'intègre ainsi dans la leçon 3 et de part sa nature probabiliste, il illustre la leçon 11.

Introduction. On considère ici un exemple où utiliser de l'aléatoire permet de vérifier une égalité plus rapidement que les algorithmes déterministes. L'idée est de vérifier si un algorithme réalise bien le produit de matrice, sans pour autant utiliser un algorithme déterministe réalisant ce calcul.

On se concentrera ici sur des matrices d'entiers modulo 2, avec A et B deux matrices, et C la matrice à tester.

Théorème 25.1 *Soient $A, B, C \in \{0,1\}^{n\times n}$. Si $AB \neq C$, et si r est choisit uniformément dans $\{0,1\}^n$, alors*

$$\mathbb{P}(ABr = Cr) \leq \frac{1}{2}$$

Pour montrer ce théorème, on utilise le lemme suivant.

Lemme 25.1 *Choisir $r = (r_1, r_2, ..., r_n) \in \{0,1\}^n$ aléatoirement et uniformément est équivalent à choisir chaque r_i uniformément et indépendamment dans $\{0,1\}$.*

Démonstration. Si chaque r_i est tiré uniformément et indépendamment, chacun des 2^n vecteurs de $\{0,1\}^n$ a une probabilité 2^{-n} d'être tiré. □

Démonstration du théorème 25.1. L'événement considéré est « $ABr = Cr$ ». Soit $D = AB - C \neq 0$. Ainsi, $ABr = Cr$ implique $Dr = 0$. Puisque $D \neq 0$, il existe au moins un coefficient non nul : on prend $d_{11} \neq 0$ sans perdre de généralité.

Puisque $Dr = 0$, on a :

$$\sum_{j=1}^{n} d_{1j} r_j = 0$$

et de manière équivalente :

$$r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}} \tag{25.1}$$

Maintenant, on utilise le lemme précédent. Au lieu de considérer qu'on tire uniformément r dans $\{0,1\}^n$, on suppose que l'on tire chaque r_i indépendamment et uniformément dans $\{0,1\}$, de r_n jusqu'à r_1. En

particulier, on peut supposer que l'on choisit $(r_2, ..., r_n)$ uniformément dans $\{0,1\}^{n-1}$ et r_1 uniformément dans $\{0,1\}$. Ces deux tirages sont réalisés de manière indépendante.

D'après la formule des probabilités totales :

$$\begin{aligned}
\mathbb{P}(ABr = Cr) &= \sum_{(x_2,...,x_n)\in\{0,1\}^{n-1}} \mathbb{P}(ABr = Cr \cap (r_2, ..., r_n) = (x_2, ..., x_n)) \\
&\leq \sum_{(x_2,...,x_n)\in\{0,1\}^{n-1}} \mathbb{P}\left(r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}} \cap (r_2, ..., r_n) = (x_2, ..., x_n)\right) \\
&= \sum_{(x_2,...,x_n)\in\{0,1\}^{n-1}} \mathbb{P}\left(r_1 = -\frac{\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}\right) . \mathbb{P}\left((r_2, ..., r_n) = (x_2, ..., x_n)\right) \\
&= \sum_{(x_2,...,x_n)\in\{0,1\}^{n-1}} \frac{1}{2} . \mathbb{P}\left((r_2, ..., r_n) = (x_2, ..., x_n)\right) \\
&= \frac{1}{2}
\end{aligned}$$

□

Pour améliorer la probabilité d'erreur du théorème précédent, il suffit de faire tourner plusieurs tests. Si l'algorithme trouve un r tel que $ABr \neq Cr$, alors l'algorithme retourne correctement $AB \neq C$. Si on trouve $ABr = Cr$ pour k tests, alors on retourne $AB = C$, ce qui est vrai avec une probabilité $\geq 1 - 2^{-k}$. La complexité de ce test est alors en $\mathcal{O}(kn^2)$.

Développement 26

Algorithme de Peterson

Auteur·e·s: Marin Malory
Références : [Tanenbaum and Bos, 2014]

Ce développement présente l'algorithme de Peterson qui permet de résoudre le problème de l'exclusion mutuelle pour deux processus, le but étant de montrer la correction de l'algorithme. Il s'insère naturellement dans les leçons qui abordent la synchronisation et la gestion de ressources, comme les leçons 13, 14 et 17. Enfin, il peut illustrer la leçon 2 si le paradigme de programmation concurrente est abordé.

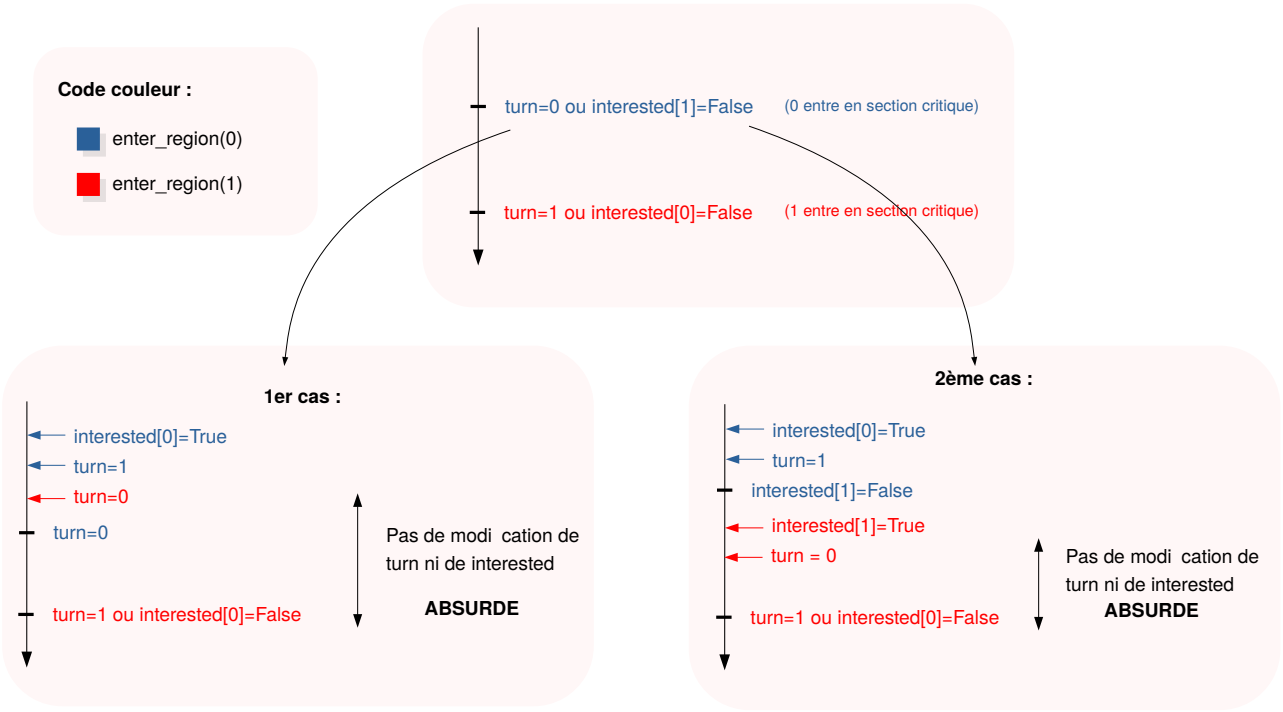
Introduction. Lorsqu'il y a une condition de concurrence, on veut que nos différents processus soient en exclusion mutuelle, c'est-à-dire qu'aucun des processus ne rentre dans leur section critique en même temps. Après une proposition non satisfaisante de Dekker (présentée par Dijkstra), Peterson a proposé une méthode élégante pour résoudre ce problème avec deux processus.

```
#define FALSE 0
#define TRUE 1
#define N 2 / * nombre de processus * /
int turn;
int interested[N]; / * à qui le tour? * /

/ * on initialise les valeurs à 0 (FALSE) * /


void enter_region(int process) / * le processus est 0 ou 1 * /
{
        int other;  /* nombre de l'autre processus */
        other = 1 - process; /* l'opposé du processus */

        interested[process] = TRUE; /* on est intéressé */
        turn = other; /* on initialise le drapeau */

         /* attente */
        while (turn == other && interested[other] == TRUE) ;
}

void leave_region(int process)
{
        /* sortie de la région critique */
        interested[process] = FALSE;
}
```

Exclusion mutuelle.

Proposition 26.1 *L'algorithme de Peterson vérifie la propriété d'exclusion mutuelle, c'est-à-dire deux processus ne rentrent jamais dans leurs sections critiques en même temps.*

Démonstration. On raisonne par l'absurde. Supposons que le processus 0 soit dans sa section critique, et le processus 1 accède à la sienne. Ainsi, `enter_region(0)` a terminé (le processus 0 a réussi à rentrer dans sa section critique) et seulement après l'appel `enter_region(1)` a terminé.



□

Exclusion mutuelle.

Proposition 26.2 *L'algorithme de Peterson ne provoque jamais d'interblocage, c'est-à-dire que si deux processus essaye d'accéder à leur section critique, au moins l'un des deux y arrive.*

Démonstration. On raisonne par l'absurde, si les deux appels sont bloqués dans la boucle `while`, on a à la fois `turn == 1` et `turn == 0` . □

Absence de famine.

Proposition 26.3 *L'algorithme de Peterson assure l'absence de famine, c'est-à-dire que tout processus qui demande à rentrer en section critique y accède en temps fini, à condition que chaque processus reste un temps fini dans sa section critique.*

Démonstration. Par l'absurde, si le processus 0 lance son appel `enter_region(0)` et reste bloque indéfiniment dans sa boucle `while` à partir de l'instant t_0. Donc pour tout temps $t \geq t_0$, on a `turn == 0 interested[1]`

`== TRUE`. Par la deuxième contrainte, on sait que le processus 1 a appelé `entre_region(1)`. De plus, le processus 1 ne reste pas bloqué, puisque sinon il y aurait interblocage. Ainsi, 1 entre en section critique et en sort en temps fini, il appelle alors `leave_region(1)`. À la fin de cet appel, on a `intersted[1] == FALSE`. Il y a alors deux cas :

— si le processus 0 reprend la main, alors on a une absurdité puisque `intersted[1] == FALSE` ;

— si le processus 1 garde et appelle à nouveau `enter_region(1)`, alors il reste cette fois bloqué dans le `while` puisqu'il est le dernier à avoir modifié `turn`. Ainsi, lorsque le processus 0 reprend la main, il est débloqué, ce qui est absurde aussi.

□

Temps d'attente borné.

Proposition 26.4 *L'algorithme de Peterson assure une attente bornée, c'est-à-dire qu'un nombre borné de processeur peuvent passer avant lui lorsqu'il demande à entrer dans sa section critique.*

Démonstration. Il suffit de reprendre la preuve précédente. □

Développement 27

Algorithme d'ordonnancement online

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Ce développement présente un simple algorithme d'ordonnancement online qui s'avère être optimal lorsqu'il s'agit de minimiser le temps de réponse maximal. Il s'insère alors naturellement dans la leçon 13 ainsi que dans toutes les leçons dans lesquelles l'ordonnanceur d'un système d'exploitation est abordé, comme les leçons 14 et 17. Enfin, puisqu'il s'agit d'un algorithme glouton, il pourra illustrer la leçon 12.

Ordonnancement online. On considère ici une stratégie simple d'ordonnancement online sur un unique processeur. Ainsi, le système reçoit les tâches au fur et à mesure et n'a pas connaissance de celles-ci à l'avance. L'idée est alors simplement d'exécuter les tâches dans leur ordre d'arrivé et montrer un certain résultat d'optimalité.

On considère une série de tâches $T_1, ..., T_n$. À chaque tâche T_i on associe :

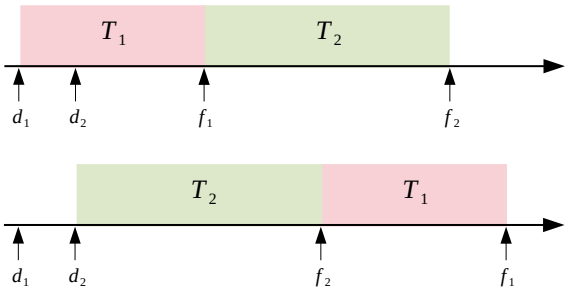
— la date d'arrivée d_i ;

— la date de fin d'exécution f_i ;

— le temps d'exécution w_i ;

— le temps de réponse $R_i = f_i - d_i$;

Remarque 27.1 *La date d'exécution (et donc le temps de réponse aussi) dépend de la stratégie adoptée par l'ordonnanceur.*



Supposons que l'on veuille minimiser le temps de réponse maximal. On considère la stratégie « Premier Arrivé Premier Servi » (PAPS) qui exécute chaque tâche dans leur ordre d'arrivé.

Théorème 27.1 *Étant donné n tâches T_i, la stratégie PAPS minimise $\max_{1\leq i\leq n} R_i$.*

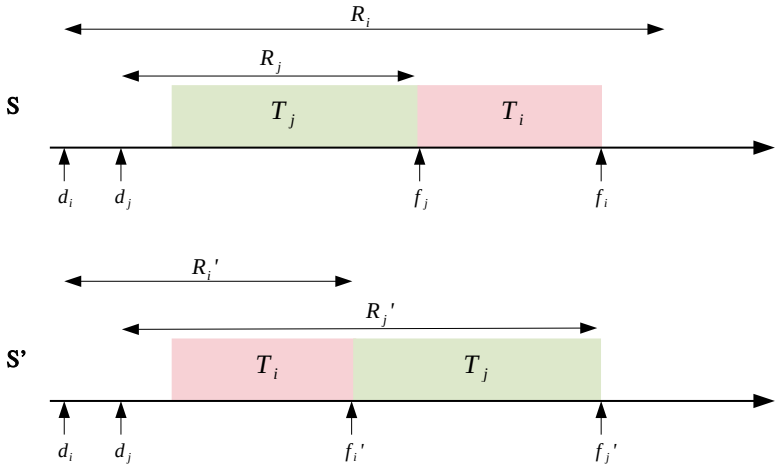
Démonstration. Le but est d'appliquer une propriété d'échange. On considère un ordonnancement optimal $\mathcal{S}$, différent de la stratégie PAPS. Ainsi, il existe deux tâches T_i et T_j telles que :

— T_i est exécutée juste après T_j (donc $f_j < f_i$) ;

— alors que T_i était disponible avant T_j ($d_i < d_j$)

Regardons ainsi les temps de réponse dans $\mathcal{S}$. On a $R_i = f_i - d_i > f_j - d_j = R_j$.

On construit alors un ordonnancement $\mathcal{S}'$ obtenu à partir de $\mathcal{S}$ en exécutant T_i avant T_j.



On remarque que les seuls temps de réponses changés sont ceux de T_i et T_j. Or, on a

$$\begin{cases} R'_i = f'_i - d_i < f_i - d_i = R_i \\ R'_j = f'_j - d_j < f_i - d_j < f_i - d_i = R_i \end{cases}$$

Ainsi, $\max(R'_i, R'_j) < R_i \leq \max(R_j, R_i)$. On en déduit alors que S' est optimal. En appliquant successivement cette propriété d'échange, on peut alors se ramener à PAPS, montrant ainsi que cette stratégie est optimale. □

Remarque 27.2 *Si on veut minimiser cette fois la somme des temps de réponses $\sum_{i=1}^{n} R_i$, PAPS n'est plus optimale.*

Développement 28

Décidabilité et langages rationnels

Auteur·e·s: Marin Malory
Références : [Sipser, 2013]

Ce développement propose d'étudier des résultats de décidabilité et d'indicibilité autour des langages rationnels, permettant d'illustrer de manière intéressante la hiérarchie de Chomsky. Il s'insère naturellement dans la leçon 27 ainsi que dans la leçon 29 si les concepts de décidabilité sont abordés.

Introduction. On considère trois problèmes autours des langages rationnels. Deux d'entre eux sont décidables, et l'autre est indécidable.

Automate et langage vide.

$$E_{AFD} = \{\langle A\rangle | A \text{ est un AFD et } L(A) = \emptyset\}$$

Théorème 28.1 *E_{AFD} est décidable.*

Démonstration. Un automate fini déterministe accepte au moins un mot si, et seulement si, il peut atteindre un état final depuis l'état initial en suivant la fonction de transition.

$\mathbf{T}$ = « Sur l'entrée $\langle A\rangle$:
1. Marquer l'état initial de A.
2. Répéter tant qu'un nouvel état est marqué :
3. Marquer chaque état qui a une transition depuis n'importe quel état marqué.
4. Si aucun état final n'est marqué, **accepter**, sinon **rejeter**. »

□

Égalité de langages rationnels. On considère le problème suivant :

$$EQ_{AFD} = \{\langle A, B\rangle | A \text{ et } B \text{ sont des AFDs et } L(A) = L(B)\}$$

Théorème 28.2 *EQ_{AFD} est décidable.*

Démonstration. On va montrer ce théorème à l'aide du précédent. On construit un nouvel automate C à partir de A et B qui accepte seulement les mots qui sont soit dans $L(A)$, soit dans $L(B)$, mais pas les deux.

Le langage de C est défini par :

$$L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$$

Remarque 28.1 *On dit que $L(C)$ est la* **différence symétrique** *de $L(A)$ et $L(B)$.*

Montrons que $L(A) = L(B)$ si, et seulement si, $L(C) = \emptyset$. Si $L(A) = L(B) = X$, alors $L(C) = (X \cap \overline{X}) \cup (\overline{X} \cap X) = \emptyset \cup \emptyset = \emptyset$. Réciproquement, si $L(C) = \emptyset$, on a $L(A) \cap \overline{L(B)} = \emptyset$ et $(\overline{L(A)} \cap L(B)) = \emptyset$. Soit $x \in L(A)$, alors par l'absurde, si $x \notin L(B)$, alors $x \in \overline{L(B)}$ et donc $x \in L(A) \cap \overline{L(B)}$, ce qui est absurde. Ainsi, on a $L(A) \subset L(B)$ et de la même manière $L(B) \subset L(A)$ ce qui conclut la preuve.

On peut alors construire C via A et B en utilisant les résultats de clôtures sur les langages rationnels. En effet, ces constructions sont des algorithmes pouvant être implantées via des machines de Turing. On en déduit la machine F qui décide EQ_{AFD} :

> $\mathbf{F}$ = « Sur l'entrée $\langle A, B \rangle$:
>
> 1. Construire C comme décrit ci-dessus.
> 2. Simuler T sur l'entrée $\langle C \rangle$.
> 3. Si T accepte, **accepter**. Si T rejette, **rejeter**. »

□

Un problème indécidable. On considère le problème suivant :

$$\mathbf{R_{TM}} = \{\langle M \rangle | M \text{ est une MT et } L(M) \text{ est rationnel}\}$$

Théorème 28.3 $\mathbf{R_{TM}}$ *est indécidable.*

Démonstration. On procède par réduction depuis le problème de l'acceptation $\mathbf{A_{TM}}$. On suppose que $\mathbf{R_{TM}}$ est décidable par une machine de Turing $\mathbf{R}$ et on l'utilise pour construire une machine de Turing $\mathbf{S}$ qui décide $\mathbf{A_{TM}}$.

Idée. $\mathbf{S}$ reçoit en entrée $\langle M, w \rangle$ où M est une machine de Turing et w un mot, et va modifier M en une nouvelle machine M_2 dont le langage est rationnel si, et seulement si M accepte w. Pour cela, M_2 va reconnaître automatique tous les mots de $\{0^n 1^n | n \geq 0\}$, mais si en plus M accepte w, alors M_2 accepte tous les autres mots.

Construction de S.

> $\mathbf{S}$ = « Sur l'entrée $\langle M, w \rangle$:
>
> 1. Construire MT M_2 suivante :
>
>> $\mathbf{M_2}$ = « Sur l'entrée x :
>>
>> (a) Si x est de la forme $0^n 1^n$, **accepter**.
>>
>> (b) Sinon, simuler M sur l'entrée w et **accepter** si M accepte w. »
>
> 2. Simuler $\mathbf{R}$ sur l'entrée $\langle M_2 \rangle$.
> 3. Si $\mathbf{R}$ accepte, **accepter** ; si $\mathbf{R}$ rejette, **rejeter**. »

Montrons que $\mathbf{S}$ décide $\mathbf{A_{TM}}$. Soit M une machine de Turing et w un mot.

Si M accepte w, alors on remarque que $L(M_2) = \Sigma^*$, car M_2 accepte sur toute entrée x. Ainsi, $L(M_2)$ est rationnel, $\mathbf{R}$ accepte sur l'entrée $\langle M_2 \rangle$ et finalement $\mathbf{S}$ accepte.

Si M n'accepte pas w, on remarque que $L(M_2) = \{0^n 1^n | n \geq 0\}$ qui n'est pas rationnel. Ainsi $\mathbf{R}$ rejette sur l'entrée $\langle M_2 \rangle$ et donc $\mathbf{S}$ rejette.

Finalement, $\mathbf{S}$ décide $\mathbf{A_{TM}}$, et ainsi $\mathbf{R_{TM}}$ est indécidable. □

Remarque 28.2 *Ce résultat nous montre qu'il n'est pas possible de décider si, lors de notre choix d'une machine de Turing comme modèle de calcul, on aurait pu simplement choisir un automate.*

Remarque 28.3 *Ce résultat est en fait un corollaire du théorème de Rice.*

Remarque 28.4 *Le fait que $\{0^n 1^n [n \geq 0\}$ n'est pas rationnel découle du lemme de l'étoile, dont on rappelle l'énoncé ci-dessous.*

Lemme 28.1 (de l'étoile) *Soit L un langage rationnel. Il existe un entier p tel que pour tout mot w de L tel que $|w| \geq p$ possède une factorisation $w = xyz$ vérifiant :*

1. *pour tout $i \geq 0, xy^i z \in L$;*
2. *$|y| > 0$;*
3. *$|xy| \leq p$.*

Développement 29

Théorème de Rice

Auteur·e·s: Marin Malory
Références : [Carton and Perrin, 2008]

Ce développement présente le théorème de Rice avec quelques applications simples du théorème. Puisque ce résultat permet de montrer l'indécidabilité de certains langages, il s'insère naturellement dans la leçon 27. De manière indirecte, ce théorème donne un résultat d'indécidabilité sur la correction des programmes et peut donc illustrer la leçon 1.

Théorème 29.1 (Théorème de Rice)
Soit P une propriété non triviale sur les langages récursivement énumérables. Le problème de savoir si le langage $L(M)$ d'une machine de Turing M vérifie P est indécidable.

De manière plus formelle, on a $P \subset \{L(M) | M$ est une machine de Turing$\}$, et on définit L_P le langage des machines de Turing vérifiant P : $L_P \subset \{< M > | L(M) \in P\}$. P est non triviale s'il existe M_1 et M_2 avec $< M_1 > \in L_P$ et $< M_2 \notin L_P$.

Exemple 29.1 *On peut prendre P la propriété : « le langage est non vide », et alors $L_P = \{< M > | L(M) \neq \emptyset\}$.*

Démonstration. On veut montrer que L_P est indécidable. On donne une réduction de $L_\in$ à L_p.

Quitte à remplacer P par sa négation, on suppose que le langage vide ne vérifie pas P. Puisque P n'est pas triviale, il existe une machine M_0 telle que $< M_0 > \in L_P$.

Pour toute paire (M, w) où M est une machine de Turing et w un mot, on définit la machine de Turing suivante :

> $\mathbf{M_w}$ = « Sur l'entrée u :
> 1. Si M accepte w, simuler M_0 sur u et **retourner le résultat**.
> 2. Sinon, **rejeter**. »

On veut montrer :

$$w \in L(M) \Leftrightarrow < M_w > \in L_P$$

Si M accepte w, alors $L(M_w) = L(M_0)$ et donc $< M_w > \in L_P$. Réciproquement, si M n'accepte pas w. Il y a deux cas :

— Si M ne s'arrête pas sur w, alors M ne s'arrête jamais et donc $L(M) = \emptyset$

— Si M rejette w, M_w rejette tous les mots u, d'où $L(M_w) = \emptyset$

Ainsi, $< M_w > \notin L_P$ puisque $\emptyset \notin L_P$.

Or, la fonction f qui à $< M, w >$ associe $< M_w >$ est calculable, et donc $L_\in$ se réduit à L_P. Ainsi, L_P est indécidable. □

Corollaire 29.1 *Le langage $L_\emptyset$ est indécidable.*

Condition d'utilisation. On considère un problème légèrement différent du problème précédent :

$$L_{\neq} = \{< M, M > | L(M) \neq L(M')\}$$

Attention, le théorème de Rice n'est pas directement applicable ici car la propriété ne porte pas sur les langages récursivement énumérables (ici sur deux langages récursivement énumérables).

On montre la proposition suivante par réduction de $L_\emptyset$ à $L_{\neq}$.

Proposition 29.1 *$L_{\neq}$ est indécidable.*

Démonstration. On fixe une machine $M_\emptyset$ qui ignore son entrée et rejette directement. On a alors $L(M_\emptyset) = \emptyset$. On pose g la fonction qui à $< M >$ associe $< M, M_\emptyset >$ (qui est calculable), et on a

$$L(M) \neq \emptyset \Leftrightarrow L(M) \neq L(M_\emptyset)$$

Ainsi, g est une réduction de $L_\emptyset$ à $L_{\neq}$, donc $L_{\neq}$ est indécidable. □

Développement 30

Codage de Huffman

Auteur·e·s: Marin Malory
Références : [Benoit et al., 2013]

Ce développement présente le codage de Huffman qui permet de compresser un texte. Il s'agit d'un exemple classique d'algorithme glouton optimal sur les textes, illustrant ainsi les leçons 12 et 9. Cette méthode servant à compresser un texte, ce développement s'insère aussi dans les leçons traitant du stockage et de l'échange de données, notamment les leçons 18 et 21. Enfin, ce codage utilise les arbres binaires de manière astucieuse, puisque chaque arbre binaire représente un code préfixe, illustrant ainsi la leçon 10. Il est à noter que pour certaines de ces leçons, un développement similaire mais moins théorique sur l'algorithme LZW pourrait être plus judicieux.

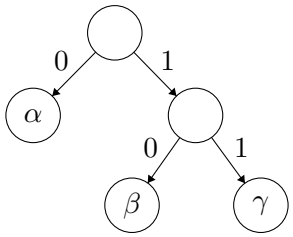
Définitions. On rappelle ici quelques définitions. On considère un alphabet Σ ayant au moins deux caractères.

Définition 30.1 (Code binaire) *On appelle* **code binaire** *toute fonction injective $c : \Sigma \to \{0,1\}^*$. L'ensemble $c(\Sigma)$ est appelé ensemble des* **mots de code**. *On peut étendre c à Σ^* par concaténation. Un code est dit* **préfixe** *si aucun mot de code n'est préfixe d'un l'autre.*

Proposition 30.1 *L'opération de décodage a une unique solution pour un code préfixe.*

On donne ici seulement une justification de la proposition ci-dessus. En effet, en lisant un mot $w \in \{0,1\}^*$, si on trouve un mot de code, alors c'est le seul possible puisqu'il n'est préfixe d'aucun autre.

Code préfixe et arbre binaire. Tout code binaire préfixe peut être représenté par un arbre binaire dont les feuilles sont les lettres de l'alphabet Σ. Un 0 est associé à chaque branche gauche et un 1 à chaque branche droite, et pour obtenir le mot de code associé à une lettre il suffit de considérer le chemin de la racine à la feuille associée. Par exemple, si l'on considère le code c sur $\Sigma = \{\alpha, \beta, \gamma\}$ défini par $c(\alpha) = 0$, $c(\beta) = 10$ et $c(\gamma) = 11$, l'arbre correspondant est le suivant :



Codage binaire optimal. On considère un texte w où chaque lettre $a \in \Sigma$ apparaît avec une fréquence $f(a)$. Étant donné un code préfixe, représenté par un arbre T, on associe un coût

$$B(T) = \sum_{a \in \Sigma} f(a) \times l_T(a)$$

où $l_T(a)$ est la taille du mot de code de a. Si $f(a)$ est exactement le nombre d'occurrences de a dans le texte w, alors $B(T)$ est exactement le nombre de bits utilisés pour encoder le texte. Un code préfixe T est dit **optimal** pour un texte si, pour ce texte, $B(T)$ est minimum.

Lemme 30.1 *Pour tout code binaire préfixe optimal, il existe un arbre binaire correspondant ayant $|\Sigma|$ feuilles et $|\Sigma| - 1$ noeuds internes.*

Démonstration. Un code préfixe optimal peut être représenté par un arbre binaire où chaque noeud interne a deux enfants. En effet, si un tel code est représenté par un arbre où un nœud interne n'a qu'un seul enfant, alors on peut supprimer ce noeud et remonter l'enfant. On obtient alors un arbre T' représentant un code préfixe avec $B(T') < B(T)$, ce qui est absurde.

Un tel arbre possède $|\Sigma|$ feuilles car il code $|\Sigma|$ lettres. Si on prend un point de vue de théorie des graphes, chaque noeud interne possède un degré 3 sauf la racine qui est de degré 2. En notant N le nombre de noeud interne et F le nombre de feuilles, on a :

$$2(N + F - 1) = 3(N - 1) + 2 + F$$

car un arbre a $N + F - 1$ arêtes. Finalement $N = F - 1 = |\Sigma| - 1$. □

Lemme 30.2 *Il existe un code préfixe optimal tel que les deux lettres de plus basses fréquences sont jumelles dans l'arbre (leurs mots de code ont la même taille mais ne diffèrent que par le dernier bit).*

Démonstration. On considère un code préfixe optimal T et on considère deux lettres de plus basses fréquences x et y. Soient a et b deux lettres qui sont jumelles dans T et à la profondeur maximale dans T. On échange x avec a et y avec b, on obtient un nouvel arbre T'. Pour montrer que T' est optimal, on montre que $B(T') \leq B(T)$. On a

$$B(T) - B(T') = f(a)(l_T(a) - l_{T'}(a)) + f(b)(l_T(b) - l_{T'}(b)) + f(x)(l_T(x) - l_{T'}(x)) + f(y)(l_T(y) - l_{T'}(y))$$

Or, on a $l_{T'}(a) = l_T(x)$ et $l_{T'}(b) = l_T(y)$, $l_T(a) = l_T(b)$ et $f(a), f(b) \leq f(x), f(y)$, et sans perdre de généralité on peut supposer $f(a) \leq f(b)$. Ainsi,

$$B(T) - B(T') \geq f(a)\left[l_T(a) - l_T(x) + l_T(a) - l_T(y)) + l_T(x) - l_T(a) + l_T(y) - l_T(a)\right] \geq 0$$

Ainsi, T' correspond à un code préfixe où les deux lettres de plus basses fréquences sont jumelles à une profondeur maximale dans l'arbre. □

La propriété précédente va définir notre choix glouton.

Lemme 30.3 *Étant donné deux lettres x et y de plus basses fréquences, on considère $\Sigma' = \Sigma \backslash \{x, y\} \cup \{z\}$ où z est une nouvelle lettre de fréquence $f(z) = f(x) + f(y)$.*

Si T' est un code optimal pour Σ', alors l'arbre T obtenu en remplaçant z par un noeud interne et deux feuilles x et y est optimal pour Σ.

Démonstration. Soit T'' un arbre optimal pour Σ, d'après le lemme précédent on peut considérer que x et y sont jumelles à profondeur maximale dans T''. En remplaçant x et y ainsi que leur parent commun par une unique feuille z, on obtient un code préfixe T''' pour Σ'.

Or, en notant $l = l_{T''}(x) = l_{T''}(y)$,

$$\begin{aligned} B(T''') &= B(T'') - f(x)l - f(y)l + (f(x) + f(y))(l-1) \\ &= B(T'') - f(x) - f(y) \end{aligned}$$

De manière similaire, on a $B(T') = B(T) - f(x) - f(y)$. Or, T' est optimal pour Σ', d'où $B(T') \leq B(T''')$ et donc $B(T) \leq B(T'')$. Ainsi, T est optimal pour Σ. □

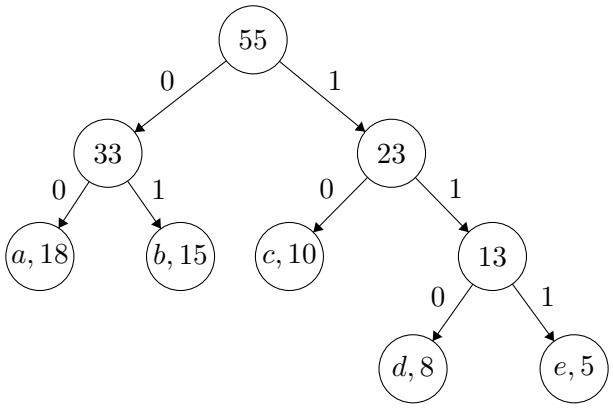
Construction du codage de Huffman. On donne ici un algorithme qui, étant donné un ensemble de fréquence par lettre, retourne un code optimal appelé **codage de Huffman**. Elle utilise une file de priorité et qui construit un arbre.

Algorithme 30.1 : CodageHuffman(Σ,f

```
F ← ConstruireFilePriorité(Σ,f) ;
n ← |Σ| ;
pour i = 1...n − 1 faire
    x ← ExtraireMin(F) ;
    y ← ExtraireMin(F) ;
    z ← NouveauNoeud() ;
    gauche[z] ← x ;
    droite[z] ← y ;
    f[z] ← f[x] + f[y] ;
    Insérer(F,z) ;
racine ← ExtraireMin(F) retourner racine
```

La complexité d'une itération de l'algorithme est un $\mathcal{O}(\log(n))$, et donc l'arbre est construit en $\mathcal{O}(n\log(n))$.

Exemple. En prenant un texte sur $\Sigma = \{a, b, c, d, e\}$ où $f(a) = 18$, $f(b) = 15$, $f(c) = 10$, $f(d) = 8$ et $f(e) = 5$ et en appliquant l'algorithme, on obtient l'arbre suivant :



et donc le code suivant : a : 00, b : 01, c : 10, d : 110 et e : 111.

Bibliographie

[Albert, 1998] Albert, L. (1998). *Cours et exercices d'informatique : classes préparatoires, 1er et 2nd cycles universitaires*. Passeport pour l'informatique. Vuibert.

[Arora and Barak, 2006] Arora, S. and Barak, B. (2006). *Computational Complexity : A Modern Approach*. Cambridge University Press.

[Beauquier et al., 2005] Beauquier, D., Berstel, J., and Chrétienne, P. (2005). *Éléments d'algorithmique*. Masson, Version.

[Belghiti et al., 2016] Belghiti, I., Mansuy, R., and Vie, J. (2016). *Les clefs pour l'info*. Im-et-Ker. Calvage et Mounet.

[Benoit et al., 2013] Benoit, A., Robert, Y., and Vivien, F. (2013). *A Guide to Algorithm Design : Paradigms, Methods, and Complexity Analysis*. Applied Algorithms and Data Structures series. Chapman & Hall/CRC.

[Biernat et al., 2015] Biernat, E., Lutz, M., and LeCun, Y. (2015). *Data science : fondamentaux et études de cas : Machine learning avec Python et R*. Eyrolles.

[Carton and Perrin, 2008] Carton, O. and Perrin, D. (2008). *Langages formels, calculabilité et complexité*. Vuibert.

[Casanova et al., 2008] Casanova, H., Legrand, A., and Robert, Y. (2008). *Parallel Algorithms*. CRC Press.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

[Crochemore and Rytter, 1994] Crochemore, M. and Rytter, W. (1994). *Text algorithms*. Maxime Crochemore.

[Dasgupta et al., 2008] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill.

[Dumas et al., 2007a] Dumas, J.-G., Roch, J.-L., Tannier, E., and Varrette, S. (2007a). *Théorie des Codes : compression, cryptage, correction*. Sciences Sup. Dunod.

[Dumas et al., 2007b] Dumas, J.-G., Roch, J.-L., Tannier, E., and Varrette, S. (2007b). *Théorie des Codes : compression, cryptage, correction*. Sciences Sup. Dunod.

[Gaudel et al., 1990] Gaudel, M.-C., Soria, M., and Froidevaux, C. (1990). *Types de données et algorithmes*. Ediscience.

[Goodrich and Tamassia, 2014] Goodrich, M. T. and Tamassia, R. (2014). *Algorithm Design and Applications*. Wiley.

[Gusfield et al., 1997] Gusfield, D., Press, C. U., and Fund, J. D. . P. S. H. L. (1997). *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. EBL-Schweitzer. Cambridge University Press.

[Hennessy and Patterson, 2012] Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition.

[Legendre, 2015] Legendre, Romain et Schwarzentruber, F. (2015). *Compilation : Analyse lexicale et syntaxique*.

[Lesesvre et al., 2020] Lesesvre, D., Barbenchon, P., Montagnon, P., and Pierron, T. (2020). *131 développements pour l'oral : agrégation externe mathématiques-informatique*. Je prépare. Dunod.

[Mitzenmacher and Upfal, 2005] Mitzenmacher, M. and Upfal, E. (2005). *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, USA.

[Motwani and Raghavan, 1995] Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

[Myers et al., 2012] Myers, G. J., Sandler, C., and Badgett, T. (2012). *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd ed edition.

[Narbel, 2005] Narbel, P. (2005). *Programmation fonctionnelle, générique et objet : une introduction avec le langage OCaml*. En pratique. Série Langages-programmation. Vuibert.

[Nielson and Nielson, 2007] Nielson, H. R. and Nielson, F. (2007). *Semantics with Applications : An Appetizer*. Undergraduate Topics in Computer Science. Springer.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational complexity.* Addison-Wesley.

[Patterson and Hennessy, 2017] Patterson, D. and Hennessy, J. (2017). *Computer Organization and Design RISC-V Edition : The Hardware Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science.

[Perifel, 2014] Perifel, S. (2014). *Complexité algorithmique*. Références sciences. Ellipses.

[Pinchinat et al., 2022] Pinchinat, S., Schwarzentruber, F., and Barbenchon, P. (2022). *Logique : fondements et applications*. Dunod.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning - From Theory to Algorithms*. Cambridge University Press.

[Silberschatz et al., 2020] Silberschatz, A., Korth, H. F., and Sudarshan, S. (2020). *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company.

[Sipser, 2013] Sipser, M. (2013). *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition.

[Stallings, 2003] Stallings, W. (2003). *Computer Organization and Architecture : Designing for Performance*. Pearson Education.

[Tanenbaum, 1976] Tanenbaum, A. (1976). *Structured Computer Organization*. Prentice-Hall series in automatic computation. Prentice-Hall.

[Tanenbaum and Bos, 2014] Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Pearson, Boston, MA, 4 edition.

[Tanenbaum and Wetherall, 2011] Tanenbaum, A. S. and Wetherall, D. (2011). *Computer Networks*. Prentice Hall, Boston, 5 edition.