Numéro d'identification :

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUES DU LANGUEDOC —

Thèse

présentée à l'Université des Sciences et Techniques du Languedoc
pour obtenir le diplôme de DOCTORAT

SPÉCIALITÉ	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Sciences pour l'Ingénieur**

Parcours en Largeur Lexicographique : Un Algorithme de Partitionnement Application aux Graphes et Généralisations

par

Christophe PAUL

Soutenue le 12 Janvier 1998 devant le Jury composé de :

M. Derek CORNEIL, Professeur, Univ. de Toronto (Canada), ... Rapporteur
M. Yves METIVIER, Professeur, Univ. de Bordeaux, LaBRi, ... Rapporteur
M. Jean-Paul BORDAT, Maître de Conférence, Univ. Montpellier II, LIRMM, ... Exanimateur
M. Yves CASEAU, Directeur de Recherches, Bouygues Telecom, ... Exanimateur
M. Michel CHEIN, Professeur, Univ. Montpellier II, LIRMM, ... Exanimateur
M. MauriceNIVAT, Professeur, Univ. Paris VII, LIAFA, ... Exanimateur
M. Bruce REED, Chargé de Recherches, CNRS Paris, ... Exanimateur
M. Michel HABIB, Professeur, Univ. Montpellier II, LIRMM, ... Directeur de Thèse

A mon grand-père,

Remerciements

Ces trois années de thèse ont été pour moi l'occasion de nombreuses rencontres toutes très enrichissantes tant sur le plan scientifique et professionnel que personnel. Je pense d'abord à Michel Habib qui a su me transmettre sa fougue et son enthousiasme pour la recherche. Je n'oublie pas non plus ses côtés humains et chaleureux exceptionnels. Quel plaisir de travailler à ses côtés dans la joie et la bonne humeur. J'espère que nous continuerons à faire un bout de route ensemble.

Derek Corneil a certainement joué, sans le savoir, un rôle très important dans cette thèse. En effet, une bonne partie des travaux présentés dans ce mémoire ont été initiés par son exposé à Marseille en septembre 1995. Pouvoir travailler avec lui et donc le rencontrer plus longuement, lors de son séjour sabbatique à Montpellier au printemps 1997, a été un des moments forts de cette thèse. Je fus très touché lorsqu'il accepta de rapporter mon travail.

La rencontre avec Yves Métivier est beaucoup plus récente. C'est à l'occasion d'un séminaire à Bordeaux en Décembre dernier que nous nous sommes vus pour la première fois. Avec une petite séance de travail, entre un repas un petit peu trop arrosé pour moi et ce séminaire, il a su me mettre à l'aise. Je lui suis très reconnaissant d'avoir accepté de faire un rapport sur ma thèse.

J'ai croisé Bruce Reed dans différents colloques. Mais c'est l'année passée, lorsqu'il m'invita pour un séjour à Princeton, que je l'ai réellement connu. Je suis très heureux de le revoir pour le jour de la soutenance. Ce jour sera d'ailleurs pour moi l'occasion de faire connaissance avec les autres membres extérieurs du jury, Yves Caseau et Maurice Nivat qui ont accepté de faire le déplacement. Bien que faisant parti du même laboratoire, je n'ai jamais travaillé avec Jean-Paul Bordat et Michel Chein. J'ai toutefois pu les apprécier lors de discussions inopportunes dans les couloirs. Je les remercie tous de s'intéresser à mes travaux et de participer au jury.

Cette thèse a aussi vu mes premiers pas dans l'enseignement. Nommé à l'IUT d'informatique de Montpellier, j'ai pu apprécié l'accueil de toute l'équipe pédagogique. Je pense en particulier à Olivier Cogis et Francis Garcia avec qui j'ai partagé la majeure partie de mes ensignements. Je n'oublie pas non plus Ehoud et Yolande Ahronovitz que j'ai su régulièrement embèter pour la mise au point de mes TD réseau et C++. En contre-partie, j'espère les avoir aidé a passer d'agréables soirées l'été dernier en les privant de leur ordinateur favori.

Une bonne partie des résultats de cette thèse sont issus d'une collaboration avec Laurent Viennot. C'est aussi au colloque de Marseille que nous nous sommes rencontrés. Depuis nous n'avons cessé de communiquer et de nous rencontrer à chaque fois avec bonne humeur. Chercheur aux multiples facettes, il sera absent de la soutenance pour cause de représentation théatrale. J'espère que nous le verrons bientôt à Monptellier ... pour une tournée nationale !

Quel bonheur d'avoir passer ces trois ans dans la zone technique ! La convivialité de tous les thésards et leur bonne humeur donnent envie de passer plus de temps au labo. Parmis eux, je tiens à saluer mes compagnons de rédaction (Vincent, Richard ...), mes relecteurs (Amit, Jean-Marc ... et Yannick !!!), qui ont su passer outre mes fôtes. Je remercie aussi toute l'équipe ALCOLO, en particulier Christian et Lhouari, pour les échanges et les discussions intéressantes et sympathiques que nous avons eu.

Je me rappellerai toujours de cette après-midi de septembre 1994 sur le trottoir du 24 rue de la Cavalerie où, un carton à la main, je me suis cogné sur un *"lampadaire à globe orange marquant un passage clouté"*. Il m'aura suivi depuis le début de mon DEA, jusqu'à la fin de cette thèse, en passant, quelque soit les conditions météo, par les plus hauts sommets Cévenols. Cela ne fait aucun doute, il aura su éclairer ma lanterne. Merci mille fois ...

Enfin, il est une autre personne qui a su supporter, non pas les intempéries, mais mes humeurs et mes égarements, en particulier ces six derniers mois. Je pense bien sûr à Mireille. Il faudra qu'elle m'explique comment elle a fait ...

Sommaire

Introduction

Une bonne idée est un moyen de rendre simple
et de résoudre un problème de manière inattendue.
R.E. Tarjan

L'algorithmique est une science que tout le monde a l'occasion de côtoyer. Pour résoudre une tâche de la vie courante, chacun d'entre nous créé son propre algorithme. Par exemple, la composition d'un itinéraire à partir d'une carte routière est ni plus ni moins un algorithme : c'est une suite d'opérations à effectuer pour atteindre une destination.

Les algorithmes sont fascinants parce que les solutions qu'ils apportent sont souvent surprenantes. Le premier qui m'ait marqué est enseigné à tous les enfants d'école primaire. Il permet de répondre à la question suivante : *un entier n est-il un multiple de 3 ?* La manière de faire ce test est très amusante, en voici une description :

Algorithme : multiple de 3

Données : un entier n
Résultat : n est-il un multiple de 3?
début
  soit S la somme de tous les chiffres de n;
  retourner (S est-il un multiple de 3?);
fin

Cette solution résume à merveille l'*algorithmique*. Certes, nous pouvions répondre à la question en posant une division et en regardant si le reste de la division est nul. Que de travail inutile ! La solution proposée est beaucoup plus astucieuse. Elle repose sur une idée simple et géniale qui permet d'éviter des calculs fastidieux. Grâce à cette méthode, le problème peut facilement se résoudre par calcul mental. Cette solution utilise implicitement la représentation décimale des entiers : sa correction en dépend, elle ne fonctionnerait pas avec une représentation binaire par exemple. La solution algorithmique dépend donc très fortement de la représentation et de la structure de données.

La plupart du temps, nous savons résoudre les problèmes de manière "brutale". Il suffit parfois d'appliquer une définition, ou lorsque nous cherchons une solution particulière (optimisant un critère de qualité par exemple), nous pouvons énumérer toutes les solutions possibles. Une telle démarche est rarement satisfaisante du point de vue du temps de calculs. Souvent efficacité et simplicité d'un algorithme riment avec astuces et idées originales qui dénotent d'ailleurs une bonne compréhension du problème.

Il est donc important de mener une étude théorique et/ou algorithmique d'un problème avant de se lancer dans la réalisation d'une solution. Une étude théorique aide à bien comprendre la structure du problème pour en dégager des propriétés qui seront la base de l'algorithme (elles en constitueront l'invariant). Ce n'est pas toujours suffisant pour obtenir une bonne efficacité (complexité théorique). Une étude "algorithmique" peut s'avérer nécessaire. Nous entendons par là, une recherche sur les structures de données à mettre en oeuvre. Ces deux aspects (études théorique et algorithmique) sont, me semble-t-il, complémentaires et indissociables. Par exemple, c'est une bonne compréhension des différentes bases de numération qui a permis de choisir la représentation décimale des nombres et donc de mettre au point la solution du test "multiple de 3".

Une structure de données astucieuse peut être à l'origine d'une complexité optimale. Mais ce souci d'efficacité peut aboutir à des structures de données très complexes. A ce stade, le travail n'est pas terminé. Il est toujours intéressant et instructif de simplifier la structure de données d'un algorithme. Même si le gain en complexité n'est pas significatif, un algorithme simple à mettre en œuvre sera préféré. Ne serait-ce que pour minimiser le risque d'erreur de programmation. Ce point est tellement vrai, que dans certains cas, nous acceptons une légère augmentation de complexité pourvu que l'algorithme en soit simplifié.

Pendant ces trois années de thèse, je me suis essentiellement intéressé à l'algorithmique de graphes. Certains des résultats que nous avons obtenus sont des simplifications d'algorithmes existants (orientation transitive, reconnaissance des graphes d'intervalles par exemple). Dans ce sens, j'espère que nous contribuons un tant soit peu à la mise au point de "beaux algorithmes".

Le titre de cette thèse indique que nous nous sommes intéressés aux *parcours en largeur lexicographiques* (en anglais, *Lexicographic Breadth First Search*), nous abrégerons Lex-BFS. A l'origine cet algorithme a été élaboré par Rose, Tarjan et Leuker [RTL76] pour reconnaître les graphes triangulés (graphes n'ayant pas de cycle sans corde de longueur strictement supérieure à 3). Depuis quelques années des propriétés intéressantes de ce parcours ont été mises en évidence. En particulier dans le cadre de la reconnaissance des graphes d'intervalles.

Le premier algorithme linéaire pour reconnaître les graphes d'intervalles est dû à Booth et Lueker [BL76]. Il utilise une structure de données compliquée : les PQ-arbres. Ce n'est que 13 ans plus tard qu'une première simplification de cet algorithme est apparue. Korte et Möhring [KM89] ont modifié les PQ-arbres grâce à l'utilisation de Lex-BFS. En 1991, Hsu et Ma [HM91] utilisent une variante de Lex-BFS pour éviter les PQ-arbres. Mais ils ont besoin d'une étape préliminaire de décomposition modulaire du graphe.

En 1991, Simon [Sim91] propose le premier algorithme que l'on peut vraiment qualifier de très simple. Son idée était d'utiliser quatre parcours Lex-BFS consécutifs. Le dernier parcours devait alors vérifier certaines propriétés si et seulement si le graphe est un graphe d'intervalles. Malheureusement, Ma [Ma] a mis en évidence un contre-exemple. Ce n'est qu'en 1997 que Corneil, Olariu et Stewart [COS97b] ont réussi à "réparer" l'algorithme de Simon. Leur preuve est encore très longue. L'algorithme de reconnaissance des graphes d'intervalles que nous présentons au chapitre 9 date de 1996. Il est aussi basé sur Lex-BFS, mais son principe est totalement différent de celui de Corneil, Olariu et Stewart.

Ces "péripéties" et avancées successives montrent combien il est difficile de bien maîtriser Lex-BFS, en particulier lorsque plusieurs parcours successifs sont utilisés. Le premier résultat obtenu dans cette direction est sans doute le double parcours Lex-BFS de Corneil, Olariu et Stewart [COS95] calculant une *paire dominante* dans les *graphes sans triplet astéroïde* .

Une *paire dominante* est une paire de sommet (x, y) telle que pour tout chemin entre x et y, un sommet quelconque z du graphe appartient à ce chemin ou possède un voisin sur ce chemin. Nous pouvons donc voir les graphes sans triplet astéroïde comme des graphes dont les sommets sont contenus dans un "cylindre" de rayon 1 autour d'un chemin entre x et y. Ces graphes ont donc une structure linéaire très forte.

Intuitivement le diamètre d'un graphe correspond à la plus grande distance (*i.e.* longueur d'un plus court chemin) entre deux sommets du graphe (des définitions précises seront données ultérieurement). Il faut remarquer que la longueur d'un plus court chemin entre deux sommets x et y d'une paire dominante diffère au plus de 2 avec le diamètre du graphe. Les paires dominantes calculées par le double parcours Lex-BFS fournissent-elles une meilleure approximation du diamètre?

De plus la classe des graphes d'intervalles est exactement l'intersection de la classe de graphes triangulés avec la classe des graphes sans triplet astéroïde. Il n'est donc pas si étonnant que Lex-BFS produise des résultats intéressant sur les graphes d'intervalles et les graphes sans triplet astéroïde. Dans quelle mesure le double Lex-BFS permet-il aussi d'approximer le diamètre d'un graphe triangulé?

Les deux questions, évoquées ci-dessus, reconnaissance des graphes d'intervalles et approximation du diamètre d'un graphe, ont servi de fil conducteur au travail présenté dans ce mémoire. Il est composé de deux parties relativement indépendantes.

La première partie débute par une étude de Lex-BFS dans son "cadre" d'origine (chapitre 1) : la reconnaissance des graphes triangulés. Ce chapitre peut être considéré comme une introduction au reste du mémoire. Nous nous intéressons ensuite au calcul du diamètre des graphes triangulés et des graphes sans triplet astéroïde.

La seconde partie est consacrée à ce que nous pensons être un nouveau paradigme de programmation intéressant : l'affinage de partition. En fait, cette technique n'est pas vraiment récente : en 1971, Hopcroft s'en servait déjà pour minimiser le nombre d'états d'un automate fini déterministe. Mais elle n'a pas été utilisée de manière systématique. Nous proposons entre autre, un algorithme de reconnaissance des graphes d'intervalles. Nous avons essayé de développer le maximum d'applications connues et quelques nouvelles en les unifiant dans une même routine de base. Cette unification a un intérêt didactique certain : elle met en évidence des points communs entre des problèmes *a priori* différents. De plus, elle débouche sur une "implémentation" très largement basée sur l'encapsulation des structures de données et la généricité (programmation par objets). L'affinage de partition a permis de simplifier certaines structures de données et de généraliser certains algorithmes existants. Nous espérons ainsi donner un aperçu des larges possibilités de cette méthode.

Finalement chacune des deux parties présente un aspect différent de la théorie des graphes : la première est plutôt combinatoire avec la preuve sur le diamètre, alors que la seconde est plutôt de nature algorithmique. Ce travail a pour l'instant fait l'objet de trois publications. La première concerne le chapitre 1, les deux autres les algorithmes de partitionnement. Seule la partie sur le diamètre (chapitre 2 et 3), trop récente, n'a pas encore été soumise à la critique.

- *"Chordal Graphs and their Clique Graph"*
  P. Galinier, M. Habib et C. Paul dans les actes de la conférence :
  Graph-Theoretic Concepts in Computer Science WG'95 (LNCS 1017)

- *"Lex-BFS and Partition Refinement, with application to transitive orientation, interval graph recognition and consecutive ones testing"*
  M. Habib, Ross Mc Connell, C. Paul et L. Viennot dans la revue :
  Theoritical Computer Science à paraître

- *"A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata"*
  M. Habib, C. Paul et L. Viennot à paraître dans les actes de :
  Symposium of Theoretical Aspect of Computer Science STACS'98

Première partie

Parcours en largeur lexicographique (Lex-BFS)

Chapitre 1

Reconnaissance des graphes triangulés

Dans les années 70-80, les parcours en profondeur et en largeur ont été longuement étudiés. Ils se sont avérés très utiles pour certaines applications : par exemple, le calcul des composantes fortements connexes d'un graphe et le test de planarité pour le parcours en profondeur. L'algorithme *Lexicographic Breadth First Search* (Lex-BFS dans la suite) est une variante du parcours en largeur classique. Lex-BFS a été introduit par Rose, Tarjan et Leuker [RTL76] pour reconnaître les graphes triangulés. Ce parcours aussi est utilisé dans différentes applications. Nous avons déjà mentionné la reconnaissance des graphe d'intervalles [KM89, HM91, COS97b], et le calcul d'une paire dominante dans les graphes sans triplet astéroïde. Il faut également citer les travaux de Dragan, Nicolai et Brandstädt [DNB96] sur les puissances d'un graphe et les parcours Lex-BFS. Dans la seconde partie de ce mémoire, nous montrons que Lex-BFS joue un rôle important dans les algorithmes de décomposition modulaire et d'orientation transitive.

Notre étude de Lex-BFS débute par son application d'origine : la reconnaissance des graphes triangulés. Ce sont les graphes n'ayant pas de cycle sans corde de longueur supérieure à 3 (*i.e.* tous les cycles minimaux sont des triangles). Ils possèdent une structure arborescente très forte. Cette classe de graphes a été très étudiée, elle intervient dans différents domaines : le calcul matriciel et les bases de données [Pey86, BP92], par exemple. De plus, c'est l'une des premières classes découvertes de graphes parfaits au sens de Berge [Ber60].

Il faut rappeler qu'en 1984, Tarjan et Yannakakis ont proposé une simplification de Lex-BFS : *Maximum Cardinality Search* (MCS par la suite). Ces deux algorithmes sont basés sur le même principe : ils construisent un *ordre d'élimina-*

tion simplicial si et seulement si le graphe est triangulé. L'existence d'un tel ordre (nous le définirons au chapitre 1) caractérise les graphes triangulés. Ces deux algorithmes sont gloutons et très simples. De ce point de vue, ils paraissent assez "magiques". Nous avons essayé de répondre à cette question : *pouvons-nous donner une preuve simple des deux algorithmes linéaires de reconnaissance des graphes triangulés ?*

En guise de réponse, nous proposons une preuve unifiée de Lex-BFS et MCS basée sur les *arbres de cliques*. Un graphe triangulé est caractérisé par un arrangement en arbre de ses cliques maximales tel que les cliques contenant un sommet donné soient connexes. Cet arbre est appelé arbre de cliques. Nous montrons qu'un arbre de cliques est un arbre recouvrant de poids maximum d'un certain graphe. Lex-BFS et MCS sont alors comparés à un algorithme de calcul d'un arbre recouvrant de poids maximum. En particulier, nous retrouvons une constatation faite par Blair et Peyton [BP92] : à chaque exécution de MCS correspond une exécution de l'algorithme classique de Prim [Pri57]. Ce résultat est aussi à rapprocher de ceux obtenu par McKee [McK]. Il y a deux conséquences directes à cette nouvelle preuve. Tout d'abord un nouvel algorithme de reconnaissance des graphes triangulés qui est une généralisation de Lex-BFS et MCS : sa preuve implique celles de Lex-BFS et MCS. En fait Lex-BFS et MCS sont deux implémentations linéaires possibles de cet algorithme. La seconde conséquence est un algorithme linéaire de construction d'un arbre de cliques. Cet algorithme sera utilisé pour la reconnaissance des graphes d'intervalles (voir chapitre 9).

1.1 Quelques résultats connus

Dans cette section, nous présentons les deux algorithmes linéaires de reconnaissance des graphes triangulés : Lex-BFS [RTL76] et MCS [TY84]. Nous rappelons simplement les résultats connus sur lesquels ils sont basés et les liens entre les graphes triangulés et les arbres sont ensuite mis en évidence. Ces relations sont utiles à la preuve proposée de Lex-BFS et MCS.

Remarque 1.1 *Les graphes que nous considérerons sont tous* ***connexes****.*

1.1.1 Propriétés et algorithmes de reconnaissance

Les graphes triangulés sont apparus sous différents noms au début des années 60. Dans la littérature anglo-saxonne, nous trouvons généralement l'appellation *"chordal graphs"* car dans tout cycle, il existe une *corde* (arête joignant deux sommets non-consécutifs du cycle). Nous utiliserons le nom *graphes triangulés* qui signifie que les cycles minimaux (au sens de l'inclusion des sommets) sont des triangles. Présentons une définition plus précise :

Définition 1.2 *[Ber60] Un graphe $G = (V, E)$ est un* ***graphe triangulé*** *si et seulement si pour tout cycle $\mu = [x_1, \ldots, x_p, x_1]$ de longueur $p > 3$, il existe une arête de E joignant deux sommets non-consécutifs de μ. Cette arête est appelée une* ***corde****.*

Remarquons tout d'abord que le graphe (la roue à 5 sommets, W_5) de la figure 1.1 n'est pas un graphe triangulé. Un exemple de graphe triangulé est présenté figure 1.2.

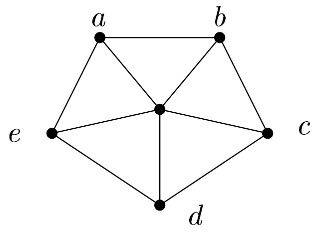


FIG. 1.1 – *La roue W_5 possède un cycle sans corde à 5 sommets : $[a, b, c, d, e, a]$.*

Notation 1.3 *Soient $G = (V, E)$ et $V' \subseteq V$ un sous-ensemble de sommets. Alors $G[V]$ désigne le sous-graphe induit par les sommets de V'*

Le premier enseignement issu de la définition des graphes triangulés est que cette classe est une classe de graphes *héréditaires.* C'est ce qu'affirme le lemme suivant :

Lemme 1.4 *Soit $G = (V, E)$ un graphe triangulé. Tout sous-graphe $H = G[V']$ de G, où $V' \subseteq V$, est un graphe triangulé .*

En effet la suppression d'un quelconque ensemble de sommet S du graphe G ne peut en aucun cas créer de cycle sans corde de longueur strictement supérieure à 3. Cette propriété d'hérédité est cruciale pour l'existence d'un ordre d'élimination des sommets.

Définition 1.5 *Soit $G = (V, E)$ un graphe :*

- *Un sommet x est* ***simplicial*** *si et seulement si son voisinage $N(x)$ est une clique.*

– *Un ordonnancement $\sigma = [x_1, \ldots, x_i, \ldots, x_n]$ est un* ***ordre d'élimination simplicial*** *si et seulement si pour tout $1 \leq i \leq n$, le sommet x_i est simplicial dans le graphe $G_i = G[x_i, \ldots, x_n]$*

Notation 1.6 *Soit $\sigma = [x_1, \ldots, x_i, \ldots, x_n]$ un ordonnancement. Dans la suite, V_i désignera toujours les sommets : $\{x_i, \ldots, x_n\}$. Le sous-graphe induit par les sommets de V_i sera noté $G_i = G[x_i, \ldots, x_n]$.*

Fulkerson et Gross [FG65] ont caractérisé les graphes triangulés de la manière suivante :

Théorème 1.7 *[FG65] Un graphe est triangulé si et seulement si il existe un ordre d'élimination simplicial.*

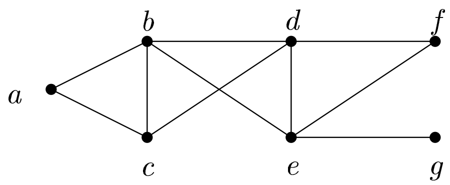


FIG. 1.2 – *$\sigma = [a, g, c, f, b, e, d]$ est un ordre d'élimination simplicial.*

Dirac a montré la propriété fondamentale suivante qui explique l'existence d'un ordre d'élimination simplicial dans les graphes triangulés.

Lemme 1.8 *[Dir61] Tout graphe triangulé $G = (V, E)$ possède un sommet simplicial. Si G n'est pas un graphe complet, alors il possède au moins 2 sommets simpliciaux non-adjacents.*

Soit G un graphe triangulé. D'après le lemme précédent, il possède un sommet simplicial x_1. Supprimons x_1, nous obtenons le sous-graphe $G_2 = G[V \setminus \{x_1\}]$. Puisque la classe des graphes triangulés est héréditaire, G_2 aussi est triangulé. Donc G_2 aussi possède un sommet simplicial x_2. Nous pouvons ainsi itérativement supprimer des sommets qui sont simpliciaux dans le graphe restant. Une telle procédure itérative a été le premier algorithme proposé [FG65] de reconnaissance des graphes triangulés.

C'est toujours cette caractérisation (existence d'un ordre d'élimination simplicial) qui a mené aux deux algorithmes linéaires de reconnaissance des graphes triangulés [RTL76, TY84]. Ils construisent tous deux un ordre d'élimination simplicial à l'envers : ils déterminent d'abord le dernier sommet x_n de l'ordre et terminent par le premier sommet x_1. Pour comprendre ce fonctionnement, nous devons utiliser un peu plus finement les conséquences de la propriété de Dirac (lemme 1.8) :

Lemme 1.9 *Soit $G = (V, E)$ un graphe triangulé. Pour tout sommet $x \in V$, il existe un ordre d'élimination simplicial σ terminant par x.*

Preuve : Soit γ un ordre d'élimination simplicial de G et $\gamma(x) = i$. A partir de γ, nous allons construire un ordre d'élimination σ tel que $\sigma(n) = x$:

Si G est un graphe complet, il suffit d'échanger x et $\gamma(n)$ pour obtenir l'ordre d'élimination simplicial σ.

Si $i = n$, alors $\sigma = \gamma$. Supposons que $i \neq n$. Alors pour tout $1 \leq j < i$, prenons $\sigma(j) = \gamma(j)$. Par hypothèse x est simplicial dans le sous-graphe G_i. Or le lemme 1.8 montre qu'il existe dans G_i un sommet simplicial $y \neq x$. Donc on peut choisir $\sigma(i) = y$.

Si y n'est pas voisin de x dans G_i, le voisinage de x reste inchangé et reste donc une clique. Sinon y a été supprimé du voisinage de x. Mais tout sous-graphe d'une clique est une clique. Donc x reste simplicial dans le sous-graphe G_{i+1}.

Donc pour tout $i \leq k \leq n$, nous pouvons appliquer l'argument précédent. Ceci montre que σ peut être construit tel que $\sigma(n) = x$. $\square$

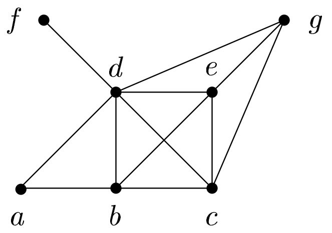
Nous pouvons donc imaginer un algorithme de reconnaissance des graphes triangulés qui commence par choisir le dernier sommet de l'ordre d'élimination simplicial. Il faut juste trouver une règle qui autorise à poursuivre cette stratégie à chaque étape sur le sous-graphe induit par les sommets non-numérotés. C'est ce que font les deux algorithmes linéaires de reconnaissance des graphes triangulés que nous allons étudier maintenant.

Le premier est dû à Rose, Tarjan et Lueker [RTL76]. C'est un parcours en largeur d'abord avec une règle de choix supplémentaire basée sur l'ordre lexicographique. Cet algorithme s'appelle (en anglais) : Lexicographic Breadth First Search. Dans la suite, nous le nommerons *Lex-BFS*.

Les sommets sont numérotés dans l'ordre inverse de leur visite. Chaque sommet x possède une marque, $marque(x)$, qui correspond à la liste ordonnée de ses voisins numérotés. A chaque fois qu'un nouveau sommet x est numéroté, son numéro est ajouté à la fin de $marque(y)$ pour tout voisin non-numéroté y de x. Le nouveau sommet choisi est toujours un sommet ayant une marque maximum dans l'ordre lexicographique.

Algorithme 1: Lex-BFS [RTL76]

Données : Un graphe $G = (V, E)$
Résultat : Un ordonnancement σ des sommets de G
début
  pour chaque *sommet* $x \in V$ **faire**
    $marque(x) \leftarrow \emptyset$;
  pour $i = n$ *à 1* **faire**
    Choisir un sommet x non numéroté de marque maximum dans l'ordre lexicographique;
    $\sigma(i) \leftarrow x$;
    pour chaque *voisin non numéroté* y *de* x **faire**
      $marque(y) \leftarrow marque(y) \cup \{i\}$;
fin



σ	7	6	5	4	3	2	1
a	$\emptyset$	7	7,6	7,6	• 7,6		
b	$\emptyset$	• 7					
c	$\emptyset$	7	• 7,6				
d	• $\emptyset$						
e	$\emptyset$	7	7,6	•7,6,5			
f	$\emptyset$	7	7	7	7	7	• 7
g	$\emptyset$	7	7	7,5	7,5,4	• 7,5,4	

FIG. 1.3 – *Un graphe triangulé G et la table d'état des marques de chaque sommet aux différentes étapes de Lex-BFS. L'ordre obtenu est $\sigma = [f, g, a, e, c, b, d]$ (il est calculé à l'envers). Nous pouvons vérifier que σ est un ordre d'élimination simplicial : par exemple e est simplicial dans le sous-graphe $G[e, c, b, d]$.*

Théorème 1.10 *[RTL76] Lex-BFS calcule un ordre d'élimination simplicial si et seulement si le graphe G est triangulé. Sa complexité est $O(n + m)$.*

Lex-BFS a ensuite été “simplifié” par Tarjan et Yannakakis [TY84]. Le choix du nouveau sommet est seulement fonction du cardinal de $marque(x)$. Il peut donc être implémenté avec un simple compteur. Par souci d'homogénéïté, nous continuerons d'utiliser $marque(x)$. Certaines preuves seront ainsi plus lisibles. L'algorithme simplifié s'appelle (en anglais) : Maximum Cardinality Search. Dans la suite, nous abrégerons par *MCS*. Cette simplification rend MCS encore plus étonnant que Lex-BFS.

Algorithme 2: MCS [TY84]

Données : Un graphe $G = (V, E)$

Résultat : Un ordonnancement σ des sommets de G

début

  pour chaque *sommet* $x \in V$ **faire**

    $marque(x) \leftarrow \emptyset$;

  pour $i = n$ *à 1* **faire**

    Choisir un sommet x non numéroté tel que $|marque(x)|$ soit maximum;

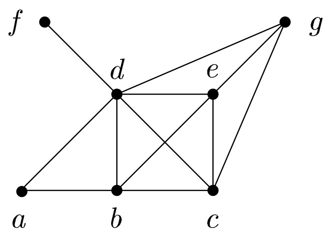
    $\sigma(i) \leftarrow x$;

    pour chaque *voisin non numéroté* y *de* x **faire**

      $marque(y) \leftarrow marque(y) \cup \{x\}$;

fin

Théorème 1.11 *[TY84] MCS calcule un ordre d'élimination simplicial si et seulement si le graphe G est triangulé. Sa complexité est $O(n + m)$.*



σ	7	6	5	4	3	2	1
a	0	1	2	2	2	• 2	
b	0	• 1					
c	0	1	• 2				
d	• 0						
e	0	1	2	• 3			
f	0	1	1	1	1	1	• 1
g	0	1	1	2	• 3		

FIG. 1.4 – *Un graphe triangulé G et la table d'état des marques de chaque sommet aux différentes étapes de MCS. L'ordre obtenu est $\sigma = [f, a, g, e, c, b, d]$ (il est calculé à l'envers). Nous pouvons vérifier que σ est un ordre d'élimination simplicial : par exemple e est simplicial dans le sous-graphe $G[e, c, b, d]$.*

Avec les exemples des figures 1.3 et 1.4, nous pouvons remarquer que les ordres d'élimination simpliciaux calculés par Lex-BFS et ceux calculés par MCS ne sont pas les mêmes. L'ordre Lex-BFS de la figure 1.3 ne peut pas être calculé par MCS et celui de la figure 1.4 ne peut pas être calculé par Lex-BFS. En effet lorsque les deux algorithmes ont numéroté les sommets $[d, b, c, e]$, le sommet de marque maximum dans Lex-BFS est a, alors que dans MCS, c'est g. La différence est que dans Lex-BFS, si la marque du sommet x à l'étape i est plus grande que celle de y, alors elle restera plus grande dans toute les étapes postérieures à i.

1.1.2 Séparateurs minimaux et arbre de cliques

Pour bien comprendre Lex-BFS et MCS, il nous reste maintenant à étudier comment ces deux algorithmes parcourent le graphe pour construire un ordre d'élimination simplicial. Pour cela, nous allons nous intéresser à une autre caractérisation des graphes triangulés : une caractérisation en termes d'arbres proposée par Gavril [Gav74] :

Théorème 1.12 *[Gav74] Un graphe est triangulé si et seulement si c'est le graphe d'intersection d'une famille de sous-arbres d'un arbre.*

Etant donné une famille $\mathcal{F}$ de sous-arbres d'un arbre $\mathcal{T}$, nous pouvons associer le graphe d'intersection G suivant : à chaque sous-arbre de $\mathcal{F}$, nous associons un sommet de G et deux sommets de G sont adjacents si et seulement si les sous-arbres correspondants partagent un nœud de $\mathcal{T}$. Cette construction est illustrée par la figure 1.5. Le théorème de Gavril nous dit que G est un graphe triangulé si et seulement si il peut être construit ainsi.

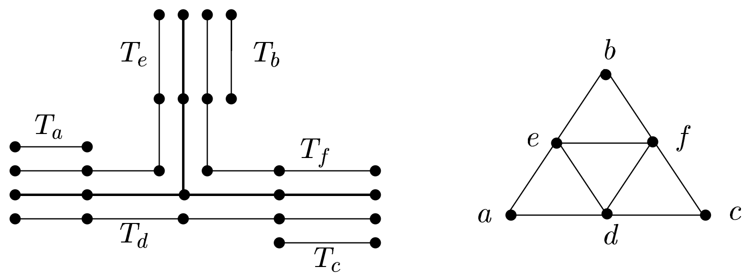


FIG. 1.5 – *Une famille de sous-arbre $\{\mathcal{T}_a, \mathcal{T}_b, \mathcal{T}_c, \mathcal{T}_d, \mathcal{T}_e, \mathcal{T}_f\}$ d'un arbre $\mathcal{T}$ et le graphe d'intersection G associé.*

Ce n'est pas toujours très simple de représenter un graphe triangulé par une famille de sous-arbres d'un arbre. Il existe une autre représentation des graphes triangulés basée sur les arbres :

Théorème 1.13 *[Gav74] Soit $G = (V, E)$ un graphe triangulé. Il existe un arbre*

$\mathcal{T} = (\mathcal{C}, \mathcal{E})$ tel que $\mathcal{C}$ est l'ensemble des cliques maximales de G; et $\forall x \in V$, l'ensemble des cliques maximales contenant x est un sous-arbre de $\mathcal{T}$. Cet arbre $\mathcal{T}$ est appelé ***arbre de cliques****.*

Comme le montre la figure 1.6, il n'existe pas un unique arbre de cliques.

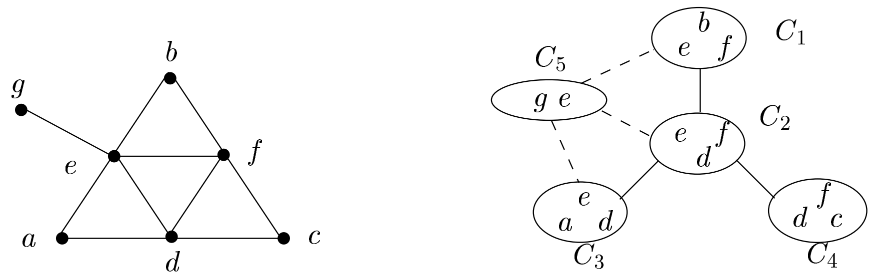


FIG. 1.6 – *Un graphe G et ses arbres de cliques. La clique C_5 doit être reliée à une des trois cliques C_1, C_2, C_3. On remarque que dans les trois cas les cliques contenant le sommet e forment un sous-arbre. Pour que l'ensemble des cliques contenant f (resp. d) soient connexes, l'arête (C_1, C_2) (resp. (C_2, C_3)) doit exister dans $\mathcal{T}$. Nous n'utilisons donc qu'une seule des arêtes en pointillés.*

Remarque 1.14 *Soit $G = (V, E)$ un graphe triangulé dont les sommets sont $x_1, \ldots, x_n$. A partir d'un arbre de cliques $\mathcal{T}$ de G, il est possible de trouver une famille $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ de sous-arbres de $\mathcal{T}$ telle que x_i et x_j sont adjacents si et seulement si $\mathcal{T}_i$ et $\mathcal{T}_j$ s'intersectent : à chaque sommet x_i, nous associons le sous-arbre $\mathcal{T}_i$ induit par l'ensemble $\mathcal{C}_i$ des cliques contenant le sommet x_i.*

- *Soient x_i et x_j deux sommets adjacents. Puisque l'ensemble des cliques maximales d'un graphe $G = (V, E)$ recouvre l'ensemble des arêtes E, il existe une clique maximale de G contenant x_i et x_j. Donc les sous-arbres $\mathcal{T}_i, \mathcal{T}_j$ ont une intersection non-vide.*

- *Réciproquement si $\mathcal{T}_i$ et $\mathcal{T}_j$ ont une intersection non vide, alors il existe une clique de G contenant les sommets x_i et x_j. Ces deux sommets sont donc adjacents.*

Les sommets d'un arbre de cliques sont clairement définis : ce sont les cliques maximales. Il n'en va pas de même des arêtes. Nous avons montré dans l'exemple de la figure 1.6, qu'il existe plusieurs arbres de cliques. A quoi correspond donc une arête d'un arbre de cliques ? Répondre à cette question est crucial pour la compréhension de la structure des graphes triangulés et en particulier de leurs algorithmes de reconnaissance. Pour cela, nous allons étudier la notion de *séparateur* d'un graphe :

Définition 1.15 *Soient $G = (V, E)$ un graphe et x, y deux sommets non adjacents. Un ensemble de sommets S est un x, y-**séparateur** si la suppression des sommets de S place x et y dans deux composantes connexes différentes du graphe $G[V \setminus S]$. S est un x, y-**séparateur minimal** s'il n'existe pas de x, y-séparateur S' strictement inclus dans S. Enfin S est un **séparateur minimal** s'il existe x et y pour lesquels S est un x, y-séparateur minimal.*

Remarque 1.16 *Si S est un séparateur minimal, alors il peut exister S' un autre séparateur minimal tel que $S' \subsetneq S$. C'est ce que montre la figure 1.7. En fait S et S' ne sont pas minimaux pour le même couple de sommets.*

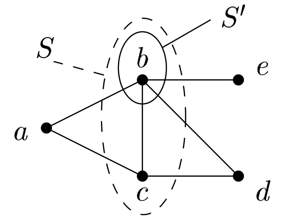


FIG. 1.7 – $S' = \{b\}$ *est un a, e-séparateur minimal,* $S = \{b, c\}$ *est un a, d-séparateur minimal.*

Dirac a montré une des premières caractérisations des graphes triangulés en termes de séparateurs minimaux :

Théorème 1.17 *[Dir61] $G = (V, E)$ est un graphe triangulé si et seulement si tout séparateur minimal est une clique.*

Si nous voulons nous faire une idée de la preuve du théorème précédent, il faut revenir un instant sur la définition des séparateurs minimaux. Un x, y-séparateur minimal S est caractérisé par l'existence pour chacun de ses sommets $s \in S$ d'un voisin dans les composantes connexes $cc(x)$ et $cc(y)$ de $G[V \setminus S]$ contenant respectivement x et y. Donc si nous prenons deux sommets s et s' dans S, nous pouvons trouver un cycle $\mu = [s, x_1, \ldots, x_k, s', y_1, \ldots, y_h, s]$ où pour tout i, $x_i \in cc(x)$ et pour tout j, $y_j \in cc(y)$. Puisque S est un séparateur, il ne peut pas y avoir de corde du type (x_i, y_j). Alors pour éviter l'existence d'un cycle sans corde de longueur strictement supérieure à 3, s et s' doivent être adjacents.

Il y a complémentarité entre les sommets des séparateurs minimaux et les sommets simpliciaux. C'est ce que nous dit le lemme suivant :

Lemme 1.18 *[Ros70] Soient $G = (V, E)$ un graphe triangulé et un sommet $x \in V$. Alors x est simplicial si et seulement si il n'appartient pas à un séparateur minimal.*

Dans son livre [Gol80], Golumbic propose en exercice de montrer que tout séparateur minimal d'un graphe triangulé est l'intersection de deux cliques maximales. Voici une autre propriété liant les arêtes d'un arbre de cliques et les séparateurs minimaux :

Lemme 1.19 *Soit $\mathcal{T}$ un arbre de cliques d'un graphe triangulé $G = (V, E)$ et soient C_a, C_b deux cliques maximales adjacentes dans $\mathcal{T}$. Alors $C_a \cap C_b$ est un séparateur minimal pour tout $a \in C_a \setminus C_b$ et $b \in C_b \setminus C_a$.*

Preuve : Faisons une preuve par l'absurde. Soient deux sommets $a \in C_a \setminus C_b$ et $x_b \in C_b \setminus C_a$. Supposons qu'il existe un chemin P_{ab} dans G entre a et b n'empruntant aucun sommet de $C_a \cap C_b$. Choisissons $P_{ab} = [a = v_0, \ldots, v_j, \ldots, v_k = x_b]$, $k \geqslant 2$, comme un plus court chemin de ce type : il ne contient donc pas de corde.

Les cliques maximales d'un graphe recouvrent l'ensemble des arêtes de ce graphe. Donc chaque arête (v_{j-1}, v_j), $0 < j \leqslant k$, de P_{ab} appartient à au-moins une clique maximale de G. Notons $\mathcal{C}(a, b)$ l'ensemble de ces cliques contenant une arête (v_{j-1}, v_j), $0 < j \leqslant k$. Comme P_{ab} est sans corde, $\forall 0 < j < k$, v_j appartient à au-moins 2 de cliques maximales de $\mathcal{C}(ab)$.

Mais $\mathcal{T}$ est un arbre où les cliques C_a et C_b sont adjacentes. Si nous supprimons l'arête (C_a, C_b) de $\mathcal{T}$, nous obtenons deux sous-arbres $\mathcal{T}_a$ et $\mathcal{T}_b$. Les cliques maximales de $\mathcal{C}(ab)$ appartiennent à l'un ou l'autre de ces sous-arbres. Il existe donc un sommet v_j, $0 < j < k$ appartenant à une clique de $\mathcal{T}_a$ et à une clique de $\mathcal{T}_b$. Or v_j n'appartient pas à $C_a \cap \mathcal{C}_b$. Donc l'ensemble des cliques contenant v_j n'est pas connexe sur $\mathcal{T}$. Cela contredit la définition des arbres de cliques.

Donc $C_a \cap C_b$ coupe tous les chemins entre a et b. De plus tout ab-séparateur doit contenir $N(a) \cap N(b)$ sinon il existe un chemin non coupé. Or $N(a) \cap N(b) = C_a \cap C_b$, c'est un séparateur minimal pour a et b. □

Corollaire 1.20 *Soient $\mathcal{T}$ un arbre de cliques du graphe triangulé $G = (V, E)$ et (C_a, C_b) une arête de $\mathcal{T}$. Notons $\mathcal{T}_a$ et $\mathcal{T}_b$ les deux sous-arbres de $\mathcal{T}$ obtenus en supprimant l'arête (C_a, C_b). Alors $C_a \cap C_b$ est un séparateur pour tout sommet $a \notin C_a \cap C_b$ d'une clique de $\mathcal{T}_a$ et $b \notin C_a \cap C_b$ d'une clique de $\mathcal{T}_b$.*

Ce corollaire est une conséquence directe du lemme précédent et de la définition d'un arbre de cliques qui exprime que les cliques contenant un sommet x donné induisent un sous-arbre. Remarquons toutefois que le séparateur $C_1 \cap C_2$ n'est pas toujours minimal pour x_1 et x_2. Dans l'exemple de la figure 1.6, supposons que C_5 soit connectée à la clique C_1 dans l'arbre de cliques. Nous remarquons alors que $C_1 \cap C_2 = \{e, f\}$ n'est pas un séparateur minimal pour g et d par exemple.

1.2 Arbres de cliques et graphe des cliques

Dans la section précédente, nous avons présenté les arbres de cliques des graphes triangulés. Cependant, nous avons vu qu'il n'y a pas unicité des arbres de cliques. Nous savons simplement qu'à toute arête d'un arbre de cliques correspond un séparateur minimal.

La réciproque n'est pas toujours vraie : si l'intersection de deux cliques maximales C_1 et C_2 est un séparateur minimal pour des sommets, $a \in C_1 \setminus C_2$ et $b \in C_2 \setminus C_1$, n'implique pas l'existence d'une arête entre C_1 et C_2 dans un arbre de cliques $\mathcal{T}$ donné (dans la figure 1.6, il n'y a qu'une seule des trois arêtes en pointillés).

Nous allons essayer de caractériser les arêtes d'un arbre de cliques. Pour cela nous introduisons la notion de graphe des cliques :

Définition 1.21 *Soit $G = (V, E)$ un graphe triangulé. Le* ***graphe des cliques*** *de G, noté $\mathcal{C}(G) = (V_c, E_c, \mu)$, est un graphe valué défini de la manière suivante :*

- *L'ensemble des sommets V_c est l'ensemble des cliques maximales de G,*
- *L'arête $(C_i, C_j) \in E_C$ si et seulement si $C_i \cap C_j$ est un séparateur minimal pour tout $a \in C_i \setminus C_j$ et $b \in C_j \setminus C_i$,*
- *Chaque arête (C_i, C_j) est valuée par $\mu(C_i, C_j) = C_i \cap C_j = S_{ij}$.*

Remarque 1.22 *Pour un graphe triangulé G, le graphe des cliques est unique.*

Notation 1.23 *Dans la suite, si C_i et C_j sont deux cliques maximales de G adjacentes dans $\mathcal{C}(G)$, nous noterons $C_i \cap C_j = S_{ij}$. Nous pourrons abusivement nommer l'arête (C_i, C_j) de $\mathcal{C}(G)$ par sa valuation S_{ij}.*

Remarque 1.24 *Soient G un graphe triangulé et $\mathcal{T}$ un arbre de cliques de G. Alors $\mathcal{T}$ est un arbre recouvrant du graphe des cliques $\mathcal{C}(G)$ (voir lemme 1.19).*

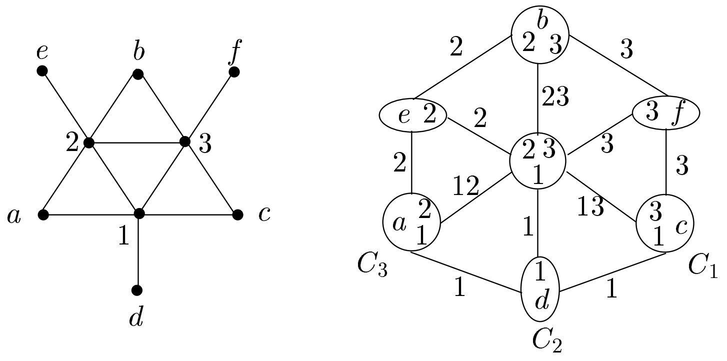


Fig. 1.8 – *Un graphe triangulé et son graphe des cliques*

Nous “caractérisons” maintenant les arbres de cliques en termes d'arbres recouvrants de poids maximum du graphe des cliques où le poids des arêtes est le cardinal de leur étiquette. Pour cela nous devons introduire une nouvelle notion :

Définition 1.25 *Soit G un graphe dont les arêtes sont étiquetées par des ensembles. Un arbre recouvrant $\mathcal{T}$ est un **arbre recouvrant inclusion-maximal** si et seulement si, pour toute arête $e = (x, y)$ n'appartenant pas à $\mathcal{T}$, l'étiquette de e ne contient strictement aucun ensemble étiquetant une arête de l'unique chemin de $\mathcal{T}$ entre x et y.*

Un tel arbre n'existe pas toujours. Cependant pour le graphes des cliques d'un graphe triangulé, il en existe un. Nous montrons qu'un arbre de poids maximum du graphe des cliques est un arbre recouvrant inclusion-maximal.

Théorème 1.26 *Soient G un graphe triangulé et $\mathcal{C}(G)$ son graphe des cliques. Soit $\mathcal{T}$ un arbre recouvrant de $\mathcal{C}(G)$. Les propriétés suivantes sont équivalentes :*

1. *$\mathcal{T}$ est un arbre de cliques*
2. *$\mathcal{T}$ est un arbre recouvrant inclusion-maximal*
3. *$\mathcal{T}$ est un arbre de poids maximum.*

Preuve : Montrons dans un premier temps que $\mathcal{T}$ est un arbre de cliques (*i.e.* les nœuds de $\mathcal{T}$ contenant un sommet donné x induisent un sous-arbre) si et seulement si $\mathcal{T}$ est un arbre couvrant de poids inclusion-maximal.

$1 \Rightarrow 2$: Supposons que $\mathcal{T}$ ne soit pas un arbre recouvrant inclusion-maximal. Alors il existe deux cliques maximales de G, C et C' adjacentes dans $\mathcal{C}(G)$ et pas dans $\mathcal{T}$, telles que le séparateur minimal $S = C \cap C'$ contient strictement un séparateur minimal $S_{i,i+1}$, $0 \leqslant i < k$, étiquetant l'arête (C_i, C_{i+1}) dans l'unique chemin $\mathcal{P} = [C = C_0, \ldots, C_i, \ldots, C_k = C']$ de $\mathcal{T}$. Soit un sommet $x \in S \setminus S_{i,i+1}$. Donc x appartient à C et C', mais $x \notin C_i \cap C_{i+1} = S_{i,i+1}$. Autrement dit l'ensemble des cliques contenant x dans $\mathcal{T}$ n'est pas connexe et ne forme pas un sous-arbre.

$2 \Leftarrow 1$: Supposons qu'il existe un sommet x tel que l'ensemble des cliques contenant x n'est pas connexe dans $\mathcal{T}$ (*i.e.* n'induit pas un sous-arbre). Nous pouvons donc choisir deux cliques C et C' non adjacentes dans $\mathcal{T}$ contenant x telles que x n'appartient à aucune clique C_i, $0 < i < k$, de l'unique chemin $\mathcal{P} = [C = C_0, \ldots, C_i, \ldots, C_k = C']$ dans $\mathcal{T}$.

Considérons le sous-graphe G' de G induit par les cliques de $\mathcal{P}_{ij}$. D'après le lemme 1.4, G' est triangulé. D'après la lemme 1.18, soit un sommet appartient à un séparateur minimal, soit il est simplicial. Comme les sommets simpliciaux n'appartiennent qu'à une seule clique, $x \in C \cap C'$ appartient donc à un séparateur minimal S. De plus x n'appartient à aucune clique de G' autre que C et C'. Or tout séparateur minimal est l'intersection de deux cliques, donc $S = C \cap C'$. Ainsi (C, C') est une arête du graphe des cliques de G' non présente dans l'arbre de $\mathcal{T}' = \mathcal{P}$. Soient $y \in C \setminus C'$, $y' \in C' \setminus C$, alors S est un y, y'-séparateur minimal.

Si pour tout i, $0 \leqslant i < k$, le séparateur minimal $S_{i,i+1}$ étiquetant l'arête $(C_i, C_{i+1}) \in \mathcal{P}$ contient un sommet t_i tel que $t_i \notin S$, alors il existe un chemin entre y et y' qui évite S : contradiction. Donc il doit exister i, $0 \leqslant i < k$, tel que $S_{i,i+1} \subseteq S$. Or S est le seul séparateur minimal à contenir x donc $S_{i,i+1} \subsetneq S$. Nous avons donc montré que $\mathcal{T}$ n'est pas un arbre recouvrant inclusion-maximal pour $\mathcal{C}(G)$.

$2 \Leftrightarrow 3$: Considérons une arête de $\mathcal{C}(G)$, (C_i, C_j) non présente dans un arbre de cliques $\mathcal{T}$. Toutes les cliques sur le chemin $\mathcal{P}_{ij}$ entre C_i et C_j contiennent les sommets de $S_{ij} = C_i \cap C_j$ (définition de l'arbre de cliques). Donc S_{ij} est inclus dans toutes les étiquettes S des arêtes de $\mathcal{P}_{ij}$. Autrement dit pour tout S, nous avons $|S_{ij}| \leqslant |S|$: $\mathcal{T}$ est un arbre couvrant de poids maximum. Comme un arbre de poids maximum est un arbre couvrant inclusion-maximal, nous avons l'équivalence.

$\square$

Nous étudions maintenant quelques propriétés des cycles du graphe des cliques. Nous pourrons alors mettre en évidence des propriétés d'échange d'arêtes pour passer d'un arbre de cliques à un autre. La figure 1.8 montre un graphe triangulé dont le graphe des cliques possède un cycle de longueur 6 sans corde : ce n'est donc pas un graphe triangulé.

Lemme 1.27 (lemme du triangle) *Soit (C_1, C_2, C_3) un cycle de $\mathcal{C}(G)$ de longueur 3. Alors parmi les trois séparateurs minimaux S_{12}, S_{23}, S_{13} correspondant aux arêtes du cycle, deux sont égaux et inclus dans le troisième.*

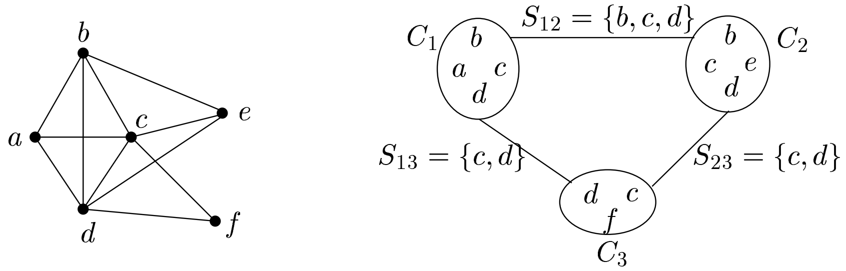


FIG. 1.9 – *Un 3-cycle dans le graphe des cliques d'un graphe triangulé. Nous avons* $S_{23} = S_{13} \subsetneq S_{12}$

Preuve : Supposons que parmi S_{12}, S_{13}, S_{23}, il existe deux séparateurs incomparables pour l'ordre d'inclusion. Choisissons S_{12} et S_{13}. Alors il existe $x \in S_{12} \setminus S_{13}$ et $y \in S_{13} \setminus S_{12}$. Autrement dit $x \in (C_1 \cap C_2) \setminus (C_1 \cap C_3)$, donc $x \notin C_2 \cap C_3$ et $y \in (C_1 \cap C_3) \setminus (C_1 \cap C_2)$, donc $y \notin C_2 \cap C_3$. Puisque C_1, C_2 et C_3 sont trois cliques maximales distinctes, $C_2 \setminus C_3$ et $C_3 \setminus C_2$ ne sont pas vides. Donc pour chaque $a \in C_2 \setminus C_3$ et $b \in C_3 \setminus C_2$, le chemin a, x, y, b n'est pas coupé par S_{23} : contradiction, par hypothèse S_{23} est un a, b-séparateur minimal dans G.

Supposons que $S_{13} \subseteq S_{23} \subseteq S_{12}$, alors $S_{13} = S_{12} \cap S_{13} = C_1 \cap C_2 \cap C_3$, et $S_{23} = S_{23} \cap S_{12} = C_1 \cap C_2 \cap C_3$. Donc nous avons $S_{13} = S_{23}$. □

Par contre, si C_1, C_2, C_3 sont trois cliques maximales de G telles que (C_1, C_2) et (C_2, C_3) sont des arêtes de $\mathcal{C}(G)$, l'arête (C_1, C_3) n'existe pas nécessairement. Il suffit pour s'en convaincre de considérer les cliques C_1, C_2, C_3 de la figure 1.8. C'est la raison pour laquelle $C(G)$ n'est pas un graphe triangulé. Nous verrons pourtant que dans certaines conditions les cycles de $\mathcal{C}(G)$ possèdent des cordes. Continuons d'abord d'étudier les cycles de longueur trois :

Lemme 1.28 *Soit $\mathcal{C}(G)$ le graphe des cliques d'un graphe triangulé G. Soient C_1, C_2, C_3 trois cliques maximales de G telles que $(C_1, C_2) \in E_c$ et $(C_2, C_3) \in E_c$, alors $S_{12} \subsetneq S_{23}$ implique $(C_1, C_3) \in E_c$.*

Preuve : Remarquons que $S_{12} = C_1 \cap C_2$ est inclus dans C_3. Soient $x_1 \in C_1 \setminus C_2$ et $x_3 \in C_3 \setminus C_2$. Si S_{12} n'est pas un x_1, x_3-séparateur, alors il existe un chemin P entre x_1 et x_3 non coupé par S_{12}. Puisque $S_{12} \subsetneq S_{23}$, il existe un sommet $y \in S_{23} \setminus S_{12}$. Notons que $y \in C_2 \setminus C_1$ et que y est adjacent à x_3. Donc le chemin P peut être prolongé jusqu'à y sans être coupé par S_{12} : S_{12} n'est donc pas un x_1, y-séparateur, contradiction. Donc S_{12} est un x_1, x_3-séparateur.

De plus $S_{12} \subseteq N(x_1) \cap N(x_3)$ et tout x_1, x_3-séparateur doit donc contenir $N(x_1) \cap N(x_3)$: donc $S_{12} = N(x_1 \cap N(x_3)$ et S_{12} est un x_1, x_3-séparateur minimal. Enfin puisque $S_{12} \subseteq C_1 \cap C_3$ et que $C_1 \cap C_3 \subseteq N(x_1) \cap N(x_3)$, nous avons aussi $S_{12} = C_1 \cap C_3$.

Montrons que S_{12} est un y_1, y_3-séparateur minimal pour tout $y_1 \in C_1 \setminus C_3$ et $y_3 \in C_3 \setminus C_1$. Puisque $C_1 \setminus C_2 = C_1 \setminus S_{12}$, alors $x_1 \in C_1 \setminus C_3$. Nous pouvons donc choisir $y_1 = x_1$. Par ailleurs $C_3 \setminus C_2 = C_3 \setminus S_{23}$, or $S_{12} \subsetneq S_{23}$, donc $C_3 \setminus S_{23} \subsetneq C_1$. Si nous choisissons $y_3 = x_3$, alors nous avons montré que S_{12} est un y_1, y_3-séparateur minimal.

Il reste à considérer le cas où $y_3 \in (C_3 \cap S_{23}) \setminus C_1 = S_{23} \setminus S_{12}$. Alors $y_3 \in C_2 \setminus C_1$, or (C_1, C_2) est une arête du graphe des cliques donc S_{12} est y_1, y_3-séparateur minimal. Nous avons donc montré que l'arête (C_1, C_3) existe dans $\mathcal{C}(G)$ et est étiquetée par S_{12}. □

Pour s'assurer de l'existence de la troisième arête lorsque les deux premières ont la même étiquette, il faut une condition supplémentaire :

Lemme 1.29 *Soient C_1, C_2, C_3 trois cliques telles que $(C_1, C_2) \in E_c$ et $(C_2, C_3) \in E_c$ et $S_{12} = S_{23}$. Si S_{12} et S_{23} sont des x_1, x_3-séparateurs pour $x_1 \in C_1 \setminus C_2$ et $x_3 \in C_3 \setminus C_2$, alors $C_1 \cap C_3$ est un séparateur minimal et l'arête (C_1, C_3) existe dans $\mathcal{C}(G)$.*

Preuve : Clairement $S_{12} \subseteq C_1 \cap C_3$ et donc $S_{23} \subseteq C_1 \cap C_3$. Montrons qu'il y a égalité. Comme tous les chemins entre x_1 et x_3 sont coupés par S_{12}, en particulier les chemins de longueur deux le sont : donc $N(x_1) \cap N(x_3) \subseteq S_{12}$. Or $C_1 \cap C_2 \subseteq N(x_1) \cap N(x_3)$, alors $S_{12} = S_{23} = C_1 \cap C_3 = N(x_1) \cap N(x_3)$. Puisque tout x_1, x_3-séparateur doit contenir $N(x_1) \cap N(x_3)$, $S_{12} = C_1 \cap C_3$ est un x_1, x_3-séparateur minimal.

De plus $x_1 \in C_1 \setminus C_2$, or $C_1 \setminus C_2 = C_1 \setminus S_{12} = C_1 \setminus C_3$, donc $x_1 \in C_1 \setminus C_3$. De même $x_3 \in C_3 \setminus C_2$ et $C_3 \setminus C_2 = C_3 \setminus S_{23} = C_3 \setminus C_1$, donc $x_3 \in C_3 \setminus C_1$. Puisque l'intersection des deux cliques est un séparateur minimal pour x_1 et x_3, l'arête (C_1, C_3) existe dans $\mathcal{C}(G)$ et est étiquetée par $S_{13} = S_{12}$. □

Lemme 1.30 (triangulation faible) *Soit $\mathcal{P}_{1k} = [C_1, \ldots, C_k]$, $k > 3$, un chemin dans un arbre de cliques $\mathcal{T}$ du graphe G. Si (C_1, C_k) est une arête de $\mathcal{C}(G)$, alors soit (C_2, C_k) soit (C_1, C_{k-1}) est une arête de $\mathcal{C}(G)$.*

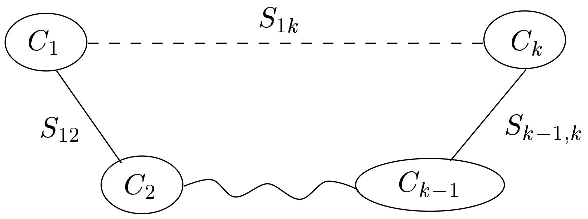


FIG. 1.10 – *Un chemin $\mathcal{P}_{1k}$ de l'arbre de cliques $\mathcal{T}$ (en trait plein) reliant deux cliques C_1 et C_k adjacentes dans $\mathcal{C}(G)$.*

Preuve : Puisque $\mathcal{T}$ est un arbre de cliques et $S_{1k} = C_1 \cap C_k$, S_{1k} est inclus dans toutes les cliques de $\mathcal{P}_{1k}$, et donc dans tout séparateur minimal $S_{i,i+1} = C_i \cap C_{i+1}$ pour $1 \leq i < k$. Donc tout $S_{i,i+1}$ est un x_1, x_k-séparateur pour $x_1 \in C_1 \setminus C_k$ et $x_k \in C_k \setminus C_1$, mais pas nécessairement minimal.

Si S_{12} (resp. $S_{k-1,k}$) contient strictement S_{1k}, alors le lemme 1.28 montre l'existence dans $\mathcal{C}(G)$ de l'arête (C_2, C_k) étiquetée par $S_{2k} = S_{1k}$ (resp. (C_1, C_{k-1}) étiquetée par $S_{1,k-1} = S_{1k}$).

Si S_{12} (resp. $S_{k-1,k}$) est égal à S_{1k}, alors le lemme 1.29 montre l'existence dans $\mathcal{C}(G)$ de l'arête (C_2, C_k) étiquetée par $S_{2k} = S_{1k}$ (resp. (C_1, C_{k-1}) étiquetée par $S_{1,k-1} = S_{1k}$). □

Les prochaines propriétés établissent une opération d'échange d'arêtes qui permet de passer d'un arbre de cliques à un autre. Pour cela, nous utilisons la propriété des 3-cycles du lemme du triangle (lemme 1.27), ainsi que le lemme de triangulation faible (lemme 1.30).

Lemme 1.31 (échange de base) *Soient $\mathcal{T}$ un arbre de cliques ayant deux arêtes (C_1, C_2) et (C_2, C_3) appartenant au même 3-cycle de $\mathcal{C}(G)$ et tels que $S_{12} \subseteq S_{23}$. Si on remplace l'arête (C_1, C_2) par la troisième arête (C_1, C_3) du cycle, nous obtenons un nouvel arbre de cliques $\mathcal{T}'$.*

Preuve : Puisque que (C_1, C_2) et (C_2, C_3) appartiennent au même 3-cycle, le lemme de triangle (lemme 1.27) implique que, parmi les séparateurs minimaux S_{12}, S_{23}, S_{13}, deux sont égaux et inclus ou égal au troisième. Puisque $\mathcal{T}$ est un arbre de cliques, S_{13} ne contient pas strictement S_{12} et S_{23} (théorème 1.26). Donc

$S_{12} = S_{13}$. Ainsi en remplaçant (C_1, C_2) par (C_1, C_3) dans $\mathcal{T}$, on obtient un nouvel arbre $\mathcal{T}'$ couvrant de même poids, donc $\mathcal{T}'$ est un arbre de poids maximum : c'est un arbre de cliques (théorème 1.26). □

L'opération décrite dans le lemme précédent sera appelée *échange de base*. Grâce à cette opération, nous pouvons montrer que le graphe des cliques est la structure minimale contenant tous les arbres de cliques.

Lemme 1.32 *Soient $\mathcal{T}$ un arbre de cliques et (C, C') une arête du graphe des cliques $\mathcal{C}(G)$ n'appartenant pas à $\mathcal{T}$. Alors il existe une arête e de $\mathcal{T}$ telle que l'arbre $\mathcal{T}'$ obtenu en échangeant e par (C, C') est un arbre de cliques.*

Preuve : Prouvons ce théorème par induction sur la longueur du chemin $\mathcal{P} = [C = C_0, \ldots, C_i, \ldots, C_k = C']$, $k \leqslant 2$, dans $\mathcal{T}$ connectant C à C'. Le cas de base (chemin de longueur 2) est prouvé par le lemme de l'échange de base (lemme 1.31). Supposons que le théorème soit vrai pour un chemin de longueur $k \geqslant 2$.

Soit $\mathcal{P} = [C = C_0, \ldots, C_i, \ldots, C_{k+1} = C']$, $k \geqslant 2$, le chemin dans $\mathcal{T}$ connectant les cliques C et C' adjacentes dans $\mathcal{C}(G)$. Le lemme de triangulation faible (lemme 1.30) montre l'existence dans $\mathcal{C}(G)$ soit de l'arête (C_0, C_k), soit de l'arête (C_1, C_{k+1}). Sans perte de généralité supposons que (C_0, C_k) existe. Donc les cliques C_0, C_k, C_{k+1} induisent un 3-cycle. Le lemme du triangle montre que deux de des séparateurs étiquetant les arêtes de ce cycle sont égaux et inclus dans le troisième. Puisque $\mathcal{T}$ est un arbre de cliques, $S_{0,k+1}$ est inclus dans S_{0k} et $S_{k,k+1}$.

1. Supposons que $S_{0,k+1} = S_{k,k+1} \subseteq S_{0k}$: échanger dans $\mathcal{T}$ l'arête (C_k, C_{k+1}) par (C_0, C_{k+1}) ne change pas le poids global de l'arbre. Nous obtenons ainsi un nouvel arbre de cliques $\mathcal{T}'$ contenant $(C_0, C_{k+1}) = (C, C')$.

2. Supposons que $S_{0,k+1} = S_{0k} \subseteq S_{k,k+1}$: alors $\mathcal{P}' = [C = C_0, \ldots, C_i, \ldots, C_k]$ est un chemin de longueur k dans $\mathcal{T}$ connectant deux cliques maximales adjacentes dans $\mathcal{C}(G)$. Par hypothèse d'induction, nous pouvons transformer $\mathcal{T}$ en un arbre de cliques $\mathcal{T}''$ en remplaçant une arête e de $\mathcal{T}$ par l'arête (C_0, C_k). Puisque $S_{0,k+1} = S_{0k}$, nous pouvons échanger dans $\mathcal{T}''$ l'arête (C_0, C_k) par (C_0, C_{k+1}), nous obtenons ainsi un arbre $\mathcal{T}'$. Finalement, pour obtenir l'arbre $\mathcal{T}'$ seule l'arête e de $\mathcal{T}$ a été échangée par (C_0, C_{k+1}).

□

Théorème 1.33 *Soient $\mathcal{T}_1$ et $\mathcal{T}_2$ deux arbres de cliques de G distincts. Alors il existe une suite finie d'échanges de base qui transforme $\mathcal{T}_1$ en $\mathcal{T}_2$.*

Preuve : Soient $\mathcal{E}_1$ les arêtes de $\mathcal{T}_1$ et $\mathcal{E}_2$ les arêtes de $\mathcal{T}_2$. Soient $e_2 = (C, C')$ une arête de $\mathcal{E}_2 \setminus \mathcal{E}_1$ et $S_2 = C \cap C'$ son étiquette. Soit $\mathcal{P}_1 = [C = C_0, \ldots, C_i, \ldots, C_k = C']$, $k > 1$, l'unique chemin dans $\mathcal{T}_1$ entre C et C'. Puisque $\mathcal{T}_1$ est un arbre de cliques, c'est un arbre recouvrant inclusion-maximal. Donc $S_2 \subseteq C_i \cap C_{i+1}$ pour tout $0 \leqslant i < k$.

Soit $\mathcal{E}'_1$ l'ensemble des arêtes de $\mathcal{P}_1$ n'appartenant pas à $\mathcal{E}_2$. Alors il existe $\mathcal{E}'_1 \neq \emptyset$, sinon $\mathcal{T}_2$ contiendrait un cycle. Puisque $\mathcal{T}_2$ est un arbre de cliques, c'est un arbre de poids maximum si nous considérons le cardinal des étiquettes. Donc il existe une arête $e_1 \in E'_1$ d'étiquette S_1 telle que $S_2 = S_1$, sinon $\mathcal{T}_2$ ne serait pas de poids maximum.

Nous pouvons donc échanger e_1 par e_2 : cela produit un nouvel arbre de cliques $\mathcal{T}'_1$ dont l'ensemble d'arêtes $E'_1 = E_1 \setminus \{e_1\} \cup \{e_2\}$. Ainsi $|E_2 \setminus E_1| > |E_2 \setminus E'_1|$. Puisque $|E_2|$ est fini, il existe une suite finie d'échanges qui transforme l'arbre de cliques $\mathcal{T}_1$ en $\mathcal{T}_2$. □

Nous retrouvons donc un résultat déjà présenté dans [HL89, Lun90]. Le multi-ensemble des séparateurs minimaux d'un arbre de cliques $\mathcal{T}$ est l'ensemble des séparateurs étiquetant les arêtes de $\mathcal{T}$. C'est un multi-ensemble puisque le même séparateur peut apparaître plusieurs fois.

Corollaire 1.34 *[HL89, Lun90] Soit G un graphe triangulé. Le multi-ensemble des séparateurs minimaux correspondant à un arbre de cliques est le même pour tous les arbres de cliques.*

Grâce au graphe des cliques, nous avons été capable de caractériser les arêtes d'un arbre de cliques. Cette structure est aussi intéressante du fait que c'est le plus petit graphe contenant tous les arbres de cliques. Le théorème 1.33 montre que l'ensemble des arbres recouvrants de poids maximum de $\mathcal{C}(G)$ a une structure bien particulière.

1.3 L'aspect glouton des algorithmes de reconnaissance

Toutes les propriétés vues dans la section précédente font penser aux matroïdes [KLS91] : un arbre de cliques peut être considéré comme un indépendant de poids maximum (une base) du graphe des cliques (c'est un arbre recouvrant de poids maximum) et nous disposons d'une opération d'échange permettant de passer d'une base à une autre. Autre similitude, il existe des algorithmes gloutons pour reconnaître les graphes triangulés.

Ces observations vont permettre de mieux analyser le comportement de Lex-BFS et MCS et surtout de les unifier dans un algorithme plus général.

Le premier algorithme présenté calcule un arbre couvrant de poids maximum de $\mathcal{C}(G)$. Il utilise fortement les propriétés sur les poids des arêtes de $\mathcal{C}(G)$ vues dans la section précédente. Nous pourrons aussi remarquer les similitudes avec l'algorithme de Prim [Pri57] : la différence réside dans le fait que les poids des arêtes ne sont pas des entiers mais des ensembles. L'ordre sur les poids n'est donc pas un ordre total, c'est l'ordre partiel d'inclusion des étiquettes.

Algorithme 3: Arbre couvrant de poids maximum du graphe des cliques

Données : Le graphe des cliques $\mathcal{C}(G)$ d'un graphe triangulé G

Résultat : $\mathcal{T} = (\mathcal{C}, \mathcal{F})$ un arbre couvrant de poids maximum de $\mathcal{C}(G)$

début

- $j \leftarrow 1$;
- Soit C_j une clique maximale;
- $Visites \leftarrow \{C_j\}$;
- $\mathcal{F} \leftarrow \emptyset$;
- **tant que** $\exists$ *une clique maximale non visitée* **faire**
  - $j \leftarrow j + 1$;
  - Choisir une clique $C_j \notin Visites$ telle que C_j soit connectée à $Visites$ par une arête e de poids maximal (pour l'ordre d'inclusion);
  - $Visites \leftarrow Visites \cup \{C_j\}$;
  - $\mathcal{F} \leftarrow \mathcal{F} \cup \{e\}$;

fin

Théorème 1.35 *Soit $\mathcal{C}(G)$ le graphe des cliques d'un graphe triangulé, l'algorithme 3 calcule un arbre couvrant de poids maximum $\mathcal{T}$: un arbre de cliques de G.*

Preuve : Soit $\mathcal{T}_j$ l'arbre construit à l'étape j. Nous allons prouver que, pour tout j, l'arbre $\mathcal{T}_j$ peut être étendu en un arbre de cliques. La propriété est triviale pour $j = 1$. Supposons qu'elle soit vraie au rang $j - 1$ pour $j \geq 2$ et que (C_i, C_j) est l'arête choisie à l'étape j.

Il existe donc un arbre couvrant de poids maximum $\mathcal{T}$ contenant $\mathcal{T}_{j-1}$. Si (C_i, C_j) est une arête de $\mathcal{T}$, il n'y a pas de problème. Sinon, il existe dans $\mathcal{T}$ un unique chemin $\mathcal{P}_{ij}$ entre C_i et C_j.

Comme l'algorithme 3 construit l'arbre de cliques $\mathcal{T}$ en rajoutant une feuille à chaque étape et que $\mathcal{T}_{j-1}$ est un sous-arbre de $\mathcal{T}$, l'ensemble des cliques du chemin $\mathcal{P}_{ij}$ appartenant à $\mathcal{T}_{j-1}$ est connexe et contient naturellement C_i. Nous avons donc $\mathcal{P}_{ij} = [C_i, \ldots, C_h, C_k, \ldots, C_j]$ où seules les cliques comprises entre C_i et C_h appartiennent à $\mathcal{T}_{j-1}$. Notons que $h < j$ et $k > j$.

Puisque $\mathcal{T}$ est un arbre de cliques, $S_{ij} = C_i \cap C_j$ est inclus dans toutes les cliques maximales de $\mathcal{P}_{ij}$ (les cliques contenant un sommet x sont connexes dans $\mathcal{T}$). En particulier $S_{ij} \subseteq S_{hk}$. Or l'algorithme choisit une arête de poids maximal dans l'ordre d'inclusion et $h < j$ et $k > j$: $S_{hk} \subseteq S_{ij}$, donc $S_{hk} = S_{ij}$. Aussi nous pouvons remplacer dans $\mathcal{T}$ l'arête (Ch, C_k) par l'arête (C_i, C_j) pour obtenir un arbre de cliques $\mathcal{T}'$ sans modifier $\mathcal{T}_{i-1}$ et sans changer le poids global de l'arbre. $\mathcal{T}'$ est donc un arbre de poids maximum : c'est un arbre de cliques. Nous avons montré que $\mathcal{T}_j$ peut être étendu en un arbre de cliques. $\square$

L'autre aspect intéressant est que à partir de l'algorithme 3, nous sommes maintenant capables de produire un nouvel algorithme qui calcule un ordre d'élimination simplicial si et seulement si le graphe en entrée est triangulé. Cet algorithme a exactement le même schéma que Lex-BFS et MCS. Seule la règle de choix change : au lieu de prendre un sommet dont la marque est maximum dans l'ordre lexicographique (Lex-BFS) ou dont le cardinal de la marque est maximum (MCS), nous choisissons un sommet dont la marque est maximale pour l'ordre d'inclusion. Nous appellerons cet algorithme (en anglais) : Maximal Inclusion Search. Nous abrégerons par MIS.

Algorithme 4: MIS

Données : Un graphe $G = (V, E)$
Résultat : Un ordonnancement σ des sommets de G
début
  pour chaque *sommet* $x \in V$ **faire**
    marque$(x) \leftarrow \emptyset$;
  pour $i = n$ *à 1* **faire**
1   Choisir un sommet x non numéroté de marque maximale dans l'ordre d'inclusion;
    $\sigma(i) \leftarrow x$;
    pour chaque *voisin non numéroté* y *de* x **faire**
      marque$(y) \leftarrow$ *marque*$(y) \cup \{i\}$;
fin

Nous avons respecter les mêmes notations que Lex-BFS et MCS. Un ordre d'élimination $\sigma = [x_1, \ldots, x_n]$ du graphe G est calculé à l'envers, c'est-à-dire que

x_{i+1} est numéroté avant x_i. Ainsi l'étape i désigne l'étape où le sommet x_i a été numéroté et G_i désigne le sous graphe induit par les sommets $\{x_n, \ldots, x_i\}$.

Théorème 1.36 *L'algorithme 4 calcule un ordre d'élimination simplicial σ si et seulement si le graphe $G = (V, E)$ est triangulé.*

Preuve : Si $\sigma = [x_1, \ldots, x_i, \ldots, x_n]$ est un ordre d'élimination simplicial, alors le graphe G est triangulé. Supposons que G soit triangulé. Soit $V_i = \{x_i, \ldots, x_n\}$. Si S est un séparateur minimal, nous noterons $cc(S)$ et $cc'(S)$ deux composantes connexes pour lesquelles S est minimal. Rappelons que tout sommet de S possède au moins un voisin dans $cc(S)$ et dans $cc'(S)$. De plus, puisque G est triangulé, les sommets de S ont au moins un voisin commun dans $cc(S)$ et dans $cc'(S)$.

Montrons par induction qu'à toute étape i, $1 \leqslant i \leqslant n$, il n'existe pas de séparateur minimal S tel que $S \not\subseteq V_i$ et $cc(S) \cap V_i \neq \emptyset$ et $cc'(S) \cap V_i \neq \emptyset$. A l'étape $i = n$, cette propriété est vraie. En effet quelque soit S, les trois ensembles $cc(S)$, $cc'(S)$ et S sont disjoints. Donc x_n ne peut être contenu que dans l'un d'entre eux. Supposons que ce soit vrai pour toute étape $k > i$.

Si la propriété n'est pas vraie à l'étape i, alors il existe un séparateur S tel que $cc(S) \cap V_i \neq \emptyset$ et $cc'(S) \cap V_i = \{x_i\}$ et il existe $s \in S$ tel que $s \notin V_i$. Puisque à toute étape, l'ensemble V_i est connexe, $N(x_i) \cap V_i \subset S$. Comme il existe un sommet dans $cc'(S)$ qui est voisin de tous les sommets de S, nous avons aussi $marque(x_i) = S \cap V_i$. Notons que S est une clique, donc $S \cap V_i \subseteq marque(s)$. Or x_i a été choisi avant s, donc $marque(s) = S \cap V_i$. Autrement dit s n'a aucun voisin numéroté dans $cc(S)$. En particulier le voisin commun $y \in cc(S)$ à tous les sommets de S n'est pas numéroté. Alors nous avons aussi $S \cap V_i \subseteq marque(y)$. Mais x_i est choisi avant y, donc $marque(y) = S \cap V_i$.

Puisque $cc(S) \cap V_i \neq \emptyset$, $S \cap V_i \neq \emptyset$ et que V_i est connexe, il existe au moins un sommet dans $S \cap V_i$ ayant des voisins dans $cc(S) \cap V_i$. Notons $N_i(S)$ l'ensemble des sommets numérotés de $cc(S)$ ayant un voisin dans $S \cap V_i$. Soit $\mathcal{P} = [a_0 = y, \ldots, a_l]$, $l > 0$, le plus court chemin entre un sommet de $N_i(S)$ et y ($a_l \in N_i(S)$). Soit l la longueur de ce chemin.

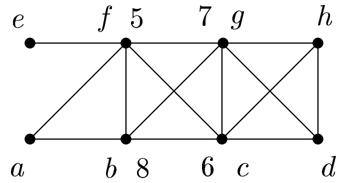
- Supposons que $l = 1$: Alors $marque(y)$ contient un sommet de $cc(S)$ et donc $marque(y) \neq S \cap V_i$. Contradiction, y devrait être numéroté avant x_i.

- Supposons que $l > 1$: a_{l-1} existe et doit être voisin de tous les sommets de $S \cap V_i$. Supposons qu'il existe $z \in S \cap V_i$ non-voisin de a_{l-1}. Soit a_i un sommet de $\mathcal{P}$ voisin de z tel que $j < l$ et j maximum. Un tel sommet existe puisque y est voisin de tous les sommets de S. Alors $\mathcal{P}' = [z, a_j, \ldots, a_{l-1}, a_l]$ est un chemin sans corde Donc pour tout $j \leqslant h < l$, le sommet a_h possède deux voisins non-adjacents. Donc a_h n'est pas simplicial. Le lemme 1.18 montre qu'il appartient à un séparateur minimal S. Ici, S doit séparer z et a_l. Puisque a_h n'est pas numéroté et que z et a_l le sont, l'hypothèse d'induction est contredite.

Donc a_{l-1} doit être voisin de tous les sommets de $S \cap V_i$. Puisque a_{l-1} est aussi voisin de $a_l \in V_i$, $(S \cap V_i) \cup \{a_l\} \subseteq marque(a_{l-1})$. Or $marque(x_i) = S \cap V_i$, a_{l-1} aurait dû être numéroté avant x_i : contradiction.

Considérons maintenant l'ordre σ. Si x_1 appartient à un séparateur minimal S, alors à l'étape $i = 2$ nous avions $S \not\subseteq V_i$ et $cc(S) \cap V_i \neq \emptyset$ et $cc'(S) \cap V_i \neq \emptyset$. Nous venons de démontrer que c'est impossible. Donc x_1 est simplicial (lemme 1.18), et il n'appartient à aucun séparateur minimal de $G_2 = G[V_2]$. L'argument peut donc à nouveau être appliqué à G_2. Nous montrons ainsi que σ est un ordre d'élimination simplicial. □

Corollaire 1.37 *Lex-BFS et MCS calculent un ordre d'élimination simplicial σ si et seulement si le graphe $G = (V, E)$ est triangulé.*



$$\begin{aligned} marque(a) &= \{8, 5\} \\ marque(e) &= \{5\} \\ marque(h) &= \{7, 6\} \\ marque(d) &= \{7, 6\} \end{aligned}$$

FIG. 1.11 – *Une exécution de l'algorithme 4. Le début de l'ordre est : b, g, c, f. Il peut continuer par h, a, d, e. Puisque $marque(e) \subsetneq marque(a)$, nous avons toujours a numéroté avant e. Par contre, a, d, h peuvent être visités dans n'importe quel ordre.*

Pour sélectionner le nouveau sommet x, il faut être capable de trouver une marque maximale dans l'ordre d'inclusion. Si nous souhaitons préserver toutes les solutions possibles, alors nous ne connaissons pas d'implémentation linéaire. Il faudrait pouvoir maintenir en $O(|N(x)|)$ l'ordre d'inclusion des marques lorsque x est numéroté.

Théorème 1.38 *Tout algorithme, basé sur l'algorithme 4 étant capable de trouver un sommet x dont la marque est maximale pour l'inclusion en temps constant, permet de calculer un ordre d'élimination simplicial en temps linéaire.*

Cependant si une marque est maximum dans l'ordre lexicographique ou de cardinal maximum, alors elle est maximale pour l'inclusion. Autrement dit, Lex-BFS et MCS sont des implémentations linéaires de MIS.

Corollaire 1.39 *Lex-BFS et MCS calculent un ordre d'élimination simplicial en temps linéaire.*

Dans [Shi84], l'auteur présente deux algorithmes qui génèrent tous les ordres d'élimination simpliciaux d'un graphe triangulé. Ce n'est pas le cas de l'algorithme 4. Par exemple pour le graphe de la figure 1.11, l'ordre d'élimination simplicial $\sigma = [d, h, g, c, b, f, e, a]$ ne peut pas être calculé (l'étiquette de e est incluse dans celle de a). Cependant le lien entre ces algorithmes d'une part et Lex-BFS et MCS d'autre part, n'était pas du tout réalisé.

1.4 Calcul d'un arbre de cliques

Nous allons montrer dans cette section comment une exécution de Lex-BFS ou de MCS permet de construire un arbre couvrant de poids maximum du graphe des cliques, c'est-à-dire un arbre de cliques. En particulier nous verrons en quoi MCS correspond à l'algorithme de Prim [Pri57].

Dans des chapitres ultérieurs (chapitres 3 et 9), nous utiliserons le calcul d'un arbre de cliques à partir de Lex-BFS. Pour éviter des redondances, nous ne proposerons qu'une adaptation de Lex-BFS. Toutefois, nous indiquerons les simplifications nécessaires pour adapter MCS en temps voulu.

L'existence d'un ordre d'élimination simplicial dans un graphe triangulé G montre que le nombre de cliques maximales de G est $O(n)$ (dans un ordre d'élimination simplicial le sommet x_i n'appartient qu'à une seule clique maximale de G_i). Mais le nombre d'arêtes du graphe des cliques est $O(n^2)$. Cette borne est atteinte par certains graphes. En vue d'obtenir un algorithme linéaire de calcul d'un arbre de cliques, il n'est donc pas possible de calculer le graphe des cliques. L'algorithme que nous proposons utilise la "trace" de Lex-BFS ou MCS pour calculer un arbre de cliques lorsque le graphe est triangulé. Cet algorithme facilite la reconnaissance des graphes triangulés. Le principe est le suivant : dans un premier temps, Lex-BFS ou MCS étendus calcule un ordre d'élimination et un arbre. Il faut ensuite vérifier que cet ordre est un ordre d'élimination simplicial. C'est le cas si et seulement si l'arbre calculé est un arbre de cliques. Par rapport aux procédures existantes ([Gol80] par exemple), le calcul de cet arbre nous permet de simplifier la vérification.

1.4.1 Trace de MCS et de Lex-BFS

Commençons par étudier l'évolution des marques des sommets au cours de Lex-BFS ou de MCS. Nous verrons ainsi en quoi ces deux algorithmes visitent consécutivement les sommets des cliques maximales d'un graphe triangulé. Nous aurons besoin dans nos preuves de distinguer la marque courante d'un sommet x, notée $marque(x)$ dans Lex-BFS et MCS, de la marque finale de x (*i.e.* la marque de x lorsque x est numéroté).

Notation 1.40 *$M(x)$ désigne la marque du sommet x à l'étape où il a été numéroté, alors que $marque(x)$ désigne la marque courante. Plus généralement dans un ordre d'élimination simplicial quelconque $\sigma = [x_1, \ldots, x_i, \ldots, x_n]$, nous avons $M(x_i) = N(x_i) \cap \{x_i, \ldots, x_n\}$.*

Lemme 1.41 *Soit σ un ordre d'élimination simplicial du graphe G. Soient x_i un sommet et $x_k \in M(x_i)$ tel que $\forall x_j \in M(x_i)$, $i > k$. Alors :*

1. *$M(x_i) \subseteq M(x_k) \cup \{x_k\}$*
2. *la clique maximale C_k de G_k contenant x_k est incluse dans la clique maximale C_i de G_i contenant x_i si et seulement si $M(x_i) = M(x_k) \cup \{x_k\}$.*

Preuve : Montrons le premier point. Supposons que $M(x_i) \not\subseteq M(x_k) \cup \{x_k\}$. Alors il existe x_j, $j > k$ tel que $x_j \notin M(x_k)$. Comme $j > k$, x_j et x_k ne sont pas adjacents. Dans ce cas, x_i ne peut pas être simplicial, contradiction : σ est un ordre d'élimination simplicial.

Montrons le second point. Puisque σ est un ordre d'élimination simplicial, pour tout j, $1 \leqslant j \leqslant n$, x_j est simplicial. Donc x_j n'appartient qu'à un seule clique maximale de G_j. Cette clique maximale est exactement $M(x_j) \cup \{x_j\}$. Donc si $M(x_k) \cup \{x_k\} = M(x_i)$, alors $C_k \subset C_i$. Si $C_k \subset C_i$, alors $M(x_k) \cup \{x_k\} \subseteq M(x_i)$. Puisque $\forall x_j \in M(x_i)$, $i > k$, le point 1 montre que $M(x_k) \cup \{x_k\} \subseteq M(x_i)$. Finalement nous avons : $M(x_k) \cup \{x_k\} = M(x_i)$. □

Définissons maintenant les notions de suite croissante de marques et séquence croissante. Elles peuvent être respectivement comparer au sous-mots et facteurs d'un mot.

Définition 1.42 *Une **suite croissante** de marques dans un ordre d'élimination simplicial σ est une suite des marques $M(x_{i_1})$, $M(x_{i_2})$, $\ldots M(x_{i_s})$, $i_1 > i_2 > \ldots > i_s$, telle que $\forall i_j$, $i_1 \geqslant i_j > i_k$, $M(x_{i_k}) = M(x_{i_{k+1}}) \cup \{x_{i_{k+1}}\}$.*

En fait, il y a une bijection entre les suites croissantes et les cliques maximales de G. Soit $U = [x_{i_1}, \ldots, x_{i_j}]$, $i_1 < \ldots < i_j$, une suite croissante, alors $U \cup M(x_i)$ est une clique maximale de G. Il suffit de se rappeler que x_{i_j} est simplicial dans G_{i_j} et que pour tout $i_k < i_j$ et $x_{i_k} \in U$, $M(x_{i_{k+1}}) = M(x_{i_k}) \cup \{x_{i_k}\}$.

Définition 1.43 *Soit σ un ordre d'élimination simplicial. Une* ***séquence croissante*** *dans σ est une suite croissante de marque dont les éléments sont consécutifs dans σ.*

Nous allons montrer que dans tout ordre d'élimination simplicial calculé par Lex-BFS ou MCS, les suites croissantes sont des séquences croissantes. Nous proposons aussi une manière de les identifier.

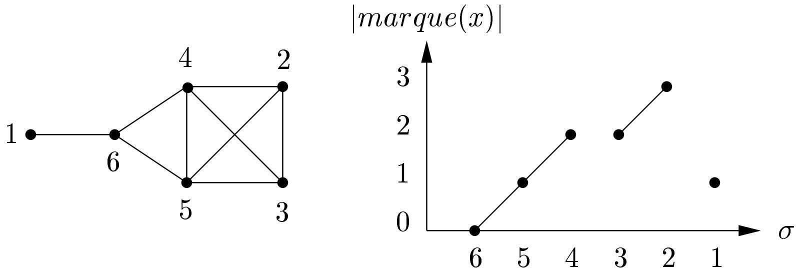


FIG. 1.12 – *Un ordre d'élimination calculé par MCS. Nous distinguons trois séquences croissantes :* $\{6, 5, 4\}$, $\{3, 2\}$ *et* $\{1\}$

Lemme 1.44 *Soit G un graphe triangulé et σ un ordre d'élimination simplicial calculé par MCS. Alors :*

1. *toute suite croissante est une séquence croissante dans σ,*
2. *les sommets x_{i+1} et x_i appartiennent à la même suite croissante si et seulement si $|M(x_{i+1})| < |M(x_i)|$.*

Preuve :

1. Soit x_i un sommet d'une suite croissante U. Puisque MCS calcule un ordre d'élimination simplicial, le prochain sommet y de U vérifie $M(y) = M(x_i) \cup \{x_i\}$. Donc avant de numéroter x_i, y et x_i avaient la même marque. Puisque le cardinal de la marque de x_i était maximum, le cardinal de la marque de y l'était aussi. De plus, le cardinal de $M(y)$ a augmenté de 1, il est désormais maximum. Le sommet y doit donc être consécutif à x_i dans σ.

2. Si x_{i+1} et x_i appartiennent à la même suite croissante, alors clairement $|M(x_{i+1})| < |M(x_i)|$. Supposons que x_i et x_{i+1} n'appartiennent pas à la même suite croissante.

   Supposons que x_{i+1} ait marqué x_i. Le point 1 du lemme 1.41 montre que $M(x_i) \subseteq M(x_{i+1}) \cup \{x_{i+1}\}$, donc il existe $y \in M(x_{i+1}) \setminus M(x_i)$. Dans ce cas nous avons $|M(x_{i+1})| \geqslant |M(x_i)|$.

   Supposons que x_{i+1} n'ait pas marqué x_i. Si $|M(x_{i+1})| < |M(x_i)|$, alors la marque de x_i n'a pas changé quand x_{i+1} a été numéroté. Autrement dit avant de numéroter x_{i+1}, nous avions $|marque(x_i)| > |marque(x_{i+1})|$: contradiction, MCS n'a pas pu choisir x_{i+1} avant x_i

□

Nous pouvons donc dire que les sommets d'une clique sont visités de manière *consécutive* : si x_i découvre une nouvelle clique (la clique maximale contenant x_i n'existait pas dans G_{i+1}), alors tous les sommets non numérotés de cette clique seront visités consécutivement à partir de x_i. MCS visite une nouvelle clique maximale si et seulement si le cardinal du sommet numéroté x_i n'est pas supérieur strictement au cardinal de la marque du sommet précédent.

Cette notion de consécutivité dans l'apparition des sommets d'une clique dans un ordre Lex-BFS existe aussi.

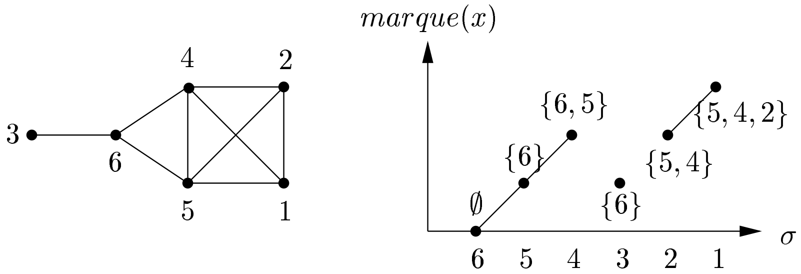


FIG. 1.13 – *Un ordre d'élimination calculé par Lex-BFS. Nous distinguons trois séquences croissantes :* $\{6, 5, 4\}$, $\{3\}$ *et* $\{2, 1\}$.

Lemme 1.45 *Soient G un graphe triangulé et σ un ordre d'élimination simplicial calculé par Lex-BFS. Alors :*

1. *toute suite croissante est une séquence croissante dans σ,*

2. *les sommets x_{i+1} et x_i appartiennent à la même suite croissante si et seulement si $|M(x_{i+1})| < |M(x_i)|$ et $x_{i+1} \in M(x_i)$.*

Preuve :

1. Soit x_i un sommet d'une suite croissante U. Puisque Lex-BFS calcule un ordre d'élimination simplicial, le prochain sommet y de U vérifie $M(y) = M(x_i) \cup \{x_i\}$. Donc avant de numéroter x_i, y et x_i avaient la même marque. Puisque $marque(x_i)$ était maximum dans l'ordre lexicographique, $marque(y)$ l'était aussi. De plus x_i a été ajouté à $marque(y)$, elle est désormais maximum dans l'ordre lexicographique. Le sommet y doit donc être consécutif à x_i dans σ.

2. Si x_{i+1} et x_i appartiennent à la même suite croissante, alors par définition $|M(x_{i+1})| < |M(x_i)|$ et $x_{i+1} \in M(x_i)$. Supposons que x_i et x_{i+1} n'appartiennent pas à la même suite croissante. Si x_{i+1} n'a pas marqué x_i le test échoue. Sinon le point 1 du lemme 1.41 montre que $M(x_i) \subseteq M(x_{i+1}) \cup \{x_{i+1}\}$, donc il existe $y \in M(x_{i+1}) \setminus M(x_i)$. Dans ce cas, $|M(x_{i+1})| \geqslant |M(x_i)|$.

□

Ainsi dans Lex-BFS pour détecter que deux sommets consécutifs x_{i+1} et x_i appartiennent à la même séquence croissante, il faut tester si $|M(x_i)| > |M(x_{k+1}|$ et $x_{i+1} \in M(x_i)$.

Intéressons-nous maintenant à la manière dont Lex-BFS et MCS passent d'une clique à l'autre. Faisons en particulier le lien avec l'algorithme 3 présenté à la section précédente.

Lemme 1.46 *A chaque exécution de Lex-BFS ou MCS sur un graphe triangulé $G = (V, E)$, nous pouvons associer une exécution de l'algorithme 3 sur le graphe des cliques $\mathcal{C}(G)$.*

Preuve : Rappelons que dans Lex-BFS et MCS, le sommet x_{i+1} est numéroté à l'étape $i + 1$ et que le premier sommet numéroté est x_n à l'étape n. Supposons qu'à l'étape $i + 1$, les cliques maximales $C_1, \ldots, C_{k-1}$ de G aient été entièrement visitées. Donc quand x_i est numéroté, nous visitons une nouvelle clique C_k. Puisque x_i est simplicial dans G_i, son voisinage est une clique. Il existe donc une clique maximale C_j avec $1 \leq j \leq k - 1$, contenant le voisinage numéroté de x_i (*i.e.* $M(x_i)$). Remarquons que $M(x_i)$ est en fait le poids de l'arête connectant C_k à C_j dans $\mathcal{C}(G)$ (en effet $M(x_i)$ est l'intersection entre C_j et C_k, c'est un séparateur minimal pour x_i et $y \in C_j \setminus C_k$). Ainsi le choix de x_i correspond au choix d'une arête dans le graphe des cliques. Si x_i a été choisi, c'est parce que $M(x_i)$ est maximum dans l'ordre lexicographique pour Lex-BFS et parce que $|M(x_i)|$ est maximum pour MCS. Dans les deux cas $M(x_i)$ est maximale pour l'ordre d'inclusion. □

En fait, un cas particulier de ce résultat était déjà connu. Dans [BP92], le lien entre MCS et l'algorithme de Prim était établi :

Corollaire 1.47 *A chaque exécution de MCS, nous pouvons associer une exécution de l'algorithme de Prim.*

1.4.2 L'algorithme

Supposons que G soit un graphe triangulé et σ un ordre d'élimination simplicial calculé par Lex-BFS. Avec le lemme 1.45, nous avons vu que les sommets de G sont partitionnés par les séquences croissantes de marques de Lex-BFS. Nous noterons Seq_j la j-ième séquence croissante de l'ordre σ. De plus, il y a bijection entre les séquences croissantes et les cliques maximales de G. Ainsi à chaque sommet x, nous associons la séquence croissante qui le contient $Seq(x)$ et donc la première clique contenant x, notée $C(x)$. D'autre part, $dernier(x)$ désigne le dernier sommet qui a marqué x. Soient x_i son premier sommet d'une séquence croissante Seq, et C la clique maximale de G correspondante, nous noterons $dernier(Seq) = dernier(C) = dernier(x_i)$.

Lemme 1.48 *Soit G un graphe triangulé. Soit $\mathcal{T}$ un arbre dont les sommets sont les cliques maximales de G et tel qu'il existe une arête entre les cliques C_j et C_k $(k < j)$ si et seulement si $dernier(C_j) \in C_k$. Alors $\mathcal{T}$ est un arbre de cliques.*

Preuve : Nous allons montrer que si deux cliques C_i et C_j contiennent un même sommet, alors toutes les cliques sur le chemin de $\mathcal{T}$ entre C_i et C_j contiennent ce sommet. Dans ce cas $\mathcal{T}$ est un arbre de cliques de G.

Soit x_i le premier sommet de la séquence croissante Seq_j associée à C_j $(C_j = C(x_i))$. Soit $y \in M(x_i)$. Montrons que toutes les cliques sur le chemin $\mathcal{P}$ dans $\mathcal{T}$ entre $C(y)$ et C_j contiennent le sommet y. Faisons une induction sur la longueur de ce chemin.

Si $\mathcal{P}$ est un chemin de longueur 1 (*i.e.* $C(y)$ et C_j sont voisines), la propriété est vraie. Supposons qu'elle le soit aussi pour un chemin de longueur $l - 1 \geq 1$.

Si la longueur de $\mathcal{P}$ est $l > 1$, alors $y \neq dernier(x_i)$ (sinon C_j et $C(y)$ seraient adjacentes). Dans le graphe G_i, le sommet x_i est simplicial. Puisque y est voisin de x_i, y et $dernier(x_i)$ sont adjacents. De plus y a été numéroté avant $dernier(x_i)$, donc $y \in M(dernier(x_i))$. Donc y appartient à la clique $C(dernier(x_i))$ qui est voisine de C_j sur $\mathcal{P}$. Alors la longueur du sous-chemin $\mathcal{P}'$ de $\mathcal{P}$ dans $\mathcal{T}$ entre $C(y)$ et $C(dernier(x_i))$ est $l - 1$ et y appartient à ces deux cliques maximales. L'hypothèse d'induction s'applique à $\mathcal{P}'$: y appartient à toutes les cliques sur $\mathcal{P}'$. Donc y appartient à toutes les cliques sur $\mathcal{P}$.

□

L'algorithme 5 est l'adaptation de Lex-BFS qui calcule un arbre de cliques en même temps qu'un ordre d'élimination simplicial. Pour MCS, il suffit à la ligne 1 de choisir une marque de cardinal maximum et à la ligne 2 de ne considérer que le cardinal des marques. S'il n'augmente pas, alors nous changeons de clique maximale (voir lemme 1.44).

Algorithme 5: Lex-BFS et calcul d'un arbre de cliques

Données : Un graphe $G = (V, E)$

Résultat : Un ordre d'élimination simplicial σ de G et un arbre de cliques $\mathcal{T}$ si et seulement si G est triangulé

début
- **pour chaque** *sommet* $x \in V$ **faire**
  - $marque(x) \leftarrow \emptyset$;
- $marquePrecedente \leftarrow \emptyset$;
- $j \leftarrow 0$;
- **pour** $i = n$ *à 1* **faire**
  - 1 Choisir un sommet x non numéroté de marque maximum dans l'ordre lexicographique;
  - 2 **si** *marquePrcdente n'est pas strictement incluse dans* $marque(x)$ **alors**
    - $j \leftarrow j + 1$;
    - Créer la clique maximale $C_j \leftarrow marque(x) \cup \{x\}$;
    - 3 $C_k = C(dernier(x))$ est le père de C_j dans $\mathcal{T}$;
    - L'arête (C_j, C_k) est étiquetée par le séparateur minimal $S_{ik} = marque(x) = C_i \cap C_k$;
  - **sinon**
    - $C_j \leftarrow C_j \cup \{x\}$;
  - 4 **pour chaque** *voisin non numéroté* y *de* x **faire**
    - $marque(y) \leftarrow marque(y) \cup \{i\}$; $dernier(y) \leftarrow x$;
  - $marquePrecedente \leftarrow marque(x)$; $\sigma(i) \leftarrow x$; $C(x) \leftarrow j$;

fin

Théorème 1.49 *L'algorithme 5 calcule un ordre d'élimination simplicial et un arbre de cliques si et seulement si le graphe G est triangulé. Sa complexité est $O(n + m)$.*

Preuve : L'existence d'un arbre de cliques caractérise les graphes triangulés, donc si l'algorithme 5 en calcule un, le graphe G est triangulé.

Le lemme 1.45, montre que les cliques maximales correspondent aux séquences croissantes. Ainsi grâce au test de la ligne 2, l'algorithme 5 construit bien l'ensemble des cliques maximales de G. Comme nous connectons toujours une clique maximale C à la clique contenant le dernier sommet qui a marqué le premier sommet de la séquence croissante de G (voir ligne 3), le lemme 1.48 montre que si G est un graphe triangulé, l'algorithme 5 construit un arbre de cliques.

Il nous reste donc à étudier le complexité de l'algorithme. A chaque fois qu'un sommet est marqué, cela correspond à une arête du graphe et chaque arête n'est utilisée qu'une fois. Donc la somme des tailles des ensembles $M(x)$ pour tous les sommets x est exactement m, le nombre d'arêtes. Il y a au plus n cliques maximales. Chaque clique maximale C_j est constituée de $Seq_j \cup M(x_i)$ où x_i est le premier sommet de Seq_j. Ainsi la somme des tailles de chacune des cliques est bornée par $O(n+m)$. Un arbre de cliques possède $O(n)$ nœuds. La taille d'un arbre de cliques est donc $O(n+m)$.

La boucle 4 se fait en la taille du voisinage de x. Il en est de même pour le test de la ligne 2, car les listes sont triées. En revanche, toutes les autres instructions de la boucle principale sont en temps constant. Donc la complexité globale est $O(n+m)$. □

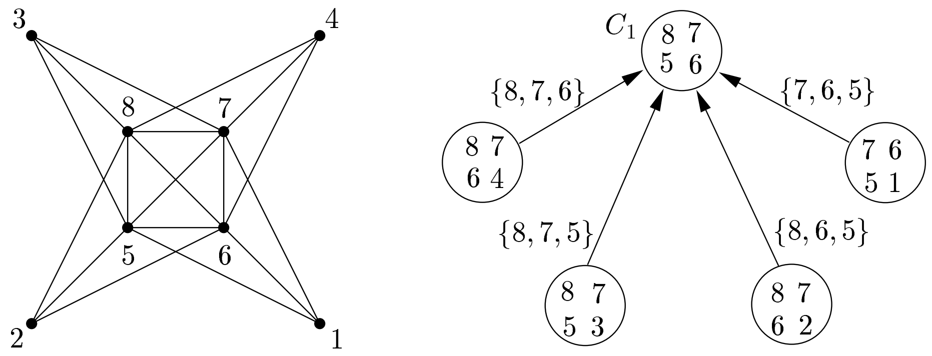
Nous expliquons maintenant comment nous pouvons tester avec l'arbre de cliques si G est un graphe triangulé. Supposons que G soit triangulé. Un sommet x_i peut avoir deux "statuts" différents :

1. x_i est le premier sommet d'une séquence croissante : alors le lemme 1.41 nous montre $M(x_i) \subseteq M(dernier(x_i)) \cup \{dernier(x_i\}$. Comme $x_i \notin C(dernier(x_i))$, nous avons $M(x_i) \subsetneq C(dernier(x_i))$.

2. x_i n'est pas le premier sommet d'une séquence croissante : alors par définition d'une séquence croissante, nous avons $M(x_i) = M(x_{i+1}) \cup \{x_{i+1}\}$.

Tester le second cas peut se faire "en ligne" (*i.e.* pendant l'algorithme 5). En effet, les marques étant des listes triées, cela demande $O(|M(x_i)|)$ opérations. Ce qui est la complexité de chaque boucle principale.

Le test correspondant au premier cas est plus délicat. En effet, si nous le faisons aussi "en ligne", il demande $O(|C(dernier(x_i)|)$. Or $|C(dernier(x_i)|$ peut être de l'ordre de $O(n)$ et l'inclusion dans cette clique $(C(dernier(x_i))$ peut aussi être fait de l'ordre de n fois. Dans ce cas, la complexité serait $O(n^2)$. Il suffit par exemple de prendre le "graphe étoile" de la figure 1.14.



$\sigma = [1, 2, 3, 4, 5, 6, 7, 8]$ est un ordre Lex-BFS

$\mathcal{T}$ l'arbre de cliques associé à σ

FIG. 1.14 – *La clique C_1 a $n/2$ sommets et elle possède $n/2$ fils dans $\mathcal{T}$.*

La vérification peut se faire globalement en $O(n + m)$ si le test est fait *a posteriori*. L'idée est que, pour une clique C donnée, il faut tester en même temps l'inclusion des marques de toutes ses cliques descendantes. Si nous supposons que le test du second cas a été effectué durant l'algorithme 5, voici un algorithme de vérification :

Algorithme 6: Reconnaissance des graphes triangulés

Données : Un arbre $\mathcal{T}$ calculé par l'algorithme 5 tel que le test 2 est effectué
Résultat : Vrai si et seulement si $\mathcal{T}$ est un arbre de cliques
début
  pour chaque *nœud* C_j *de* $\mathcal{T}$ **faire**
    $S \leftarrow$ l'ensemble des fils C_k de C_j dans $\mathcal{T}$;
1   **si** $\exists$ *une arête* (C_k, C_j) *dont l'étiquette n'est pas incluse dans* C_j **alors**
      retourner *faux*;
  retourner *vrai*;
fin

Théorème 1.50 *L'algorithme 6 vérifie si un graphe est triangulé en $O(n + m)$.*

Preuve : Nous avons vu que le test du second cas peut se faire en ligne pendant l'algorithme 5. Pour s'assurer que l'ordre d'élimination calculé par l'algorithme 5 est un ordre d'élimination simplicial, il ne reste donc plus qu'à faire le test du premier cas. C'est ce que fait l'algorithme 6 à la ligne 1.

En ce qui concerne la complexité et le détail du test de la ligne 1, nous disposons des structures de données suivantes : à chaque nœud de l'arbre, nous associons deux listes triées suivant l'ordre d'élimination calculé par l'algorithme 5, une contenant l'ensemble de ses sommets et l'autre pour la séquence croissante associée. Notons que le coût du tri de ces listes est négligeable puisque nous pouvons construire ces listes pendant l'algorithme 5. Enfin chaque sommet x dispose de sa marque $M(x)$ sous forme de liste triée.

Puisque toutes les listes sont triées, le test de la ligne 1 peut se faire en parcourant simultanément C_k et les marques des premiers sommets des séquences croissantes associées aux cliques de S. Ainsi chaque clique et chaque liste $M(x)$ est visitée au plus une fois. Puisque la taille de l'arbre des cliques est $O(n+m)$ et la somme des tailles des listes $M(x)$ est aussi $O(n+m)$, la complexité globale est $O(n+m)$. □

1.5 Conclusions

Nous avons présenté dans ce chapitre la notion de graphe des cliques d'un graphe triangulé. D'autres travaux [BP92, Pey86] ont utilisé une notion voisine : le graphe d'intersection des cliques ou l'hypergraphe des cliques. Notre définition du graphe des cliques a permis de caractériser les arbres de cliques et de comparer Lex-BFS et MCS à des algorithmes de calcul d'arbres recouvrants de poids maximum. Une généralisation de ces deux algorithmes et un algorithme linéaire de calcul d'un arbre de cliques ont alors été présentés.

Enfin ces algorithmes sont basés sur un paradigme de reconnaissance intéressant :

1. Un premier algorithme calcule une structure qui caractérise la classe de graphe en supposant que le graphe donné appartienne à cette classe.

2. Un second algorithme vérifie si la structure obtenue est bien celle escomptée.

Ce paradigme est utilisé pour la reconnaissance des graphes distances héréditaires [Dam97]. Nous l'avons aussi mis en œuvre pour les cographes, les graphes de comparabilité (voir chapitre 8) et les graphes d'intervalles (voir chapitre 9).

Chapitre 2

Diamètre d'un graphe

Ayant présenté et expliqué Lex-BFS au chapitre 1, nous consacrons les chapitres 2 et 3 à une nouvelle application de Lex-BFS : l'approximation du diamètre d'un graphe. Le diamètre d'un graphe est la plus grande distance (*i.e.* longueur d'un plus court chemin) entre deux sommets du graphe (des définitions précises seront données ultérieurement). Ce travail a été effectué en collaboration avec D. Corneil, Professeur à Toronto, lors de son séjour sabbatique à Montpellier, en 1997.

Le diamètre est un important paramètre pour la mise en place de réseaux de communications. Supposer que, les temps de communication entre deux sites donnés dépendent de la longueur du chemin les séparant est une hypothèse raisonnable. Si le réseau doit répondre à des critères de performance sur le temps de communication, nous devons alors prendre en compte son diamètre. Le concepteur peut aussi vouloir garantir une efficacité minimum en cas de panne : par exemple, si une ligne du réseau est coupée, le diamètre du réseau ne doit pas dépasser une certaine valeur [Les84]. Le diamètre est donc fortement relié aux problèmes de connectivité.

Le calcul du diamètre d'un graphe est souvent ramené au calcul des plus courts chemins (*i.e.* distances minimum) entre tout couple de sommets. Il n'est certes pas nécessaire de les calculer tous [Les84], mais dans le pire des cas le coût global est du même ordre. En utilisant un algorithme classique de calcul des plus courts chemins, la complexité du calcul du diamètre d'un graphe est $O(nm)$, où n est le nombre de sommets et m le nombre d'arêtes. Des algorithmes randomisés améliorent cette borne [MR95]. Certaines classes de graphes admettent des algorithmes *ad hoc* linéaires. C'est le cas des graphes distances héréditaires et les graphes dualement triangulés ("dually chordal graphs") [BCD94]. Nous proposons quelques résultats pour les graphes triangulés et les graphes sans triplet astéroïde.

Décrivons brièvement l'idée amusante qui est à l'origine de notre méthode. Choisissons un sommet du graphe au hasard et exécutons Lex-BFS en partant de ce sommet. Puisque Lex-BFS est un parcours en largeur, le dernier sommet visité est alors un des sommets les plus éloignés du sommet initial. Nous sommes en quelque sorte sur le "bord" du graphe. Si nous répétons cette procédure à partir du dernier sommet obtenu, dans quelle mesure Lex-BFS termine-t-il sur un sommet qui est l'extrémité d'un diamètre du graphe?

Le double parcours Lex-BFS approxime la valeur du diamètre à 1 près pour les graphes triangulés et les graphes sans triplet astéroïde. C'est donc une très bonne heuristique. Les preuves concernant ces deux classes sont très différentes. Nous ne fournirons que celles concernant les graphes triangulés au chapitre 3 (celles sur les graphes sans triplet astéroïde sont dues au Pr. D. Corneil). Il est curieux de constater que cet algorithme échoue sur les deux classes de graphes pour des raisons "opposées" : nous verrons que les contre-exemples des graphes triangulés possèdent des triplets astéroïdes et que les contre-exemples des graphes sans triplet astéroïde possèdent des cycles sans corde de longueur 4.

En utilisant la notion de *transformation polynomiale* [PS85], nous comparons ensuite le calcul du diamètre d'un graphe (quelconque) à différents problèmes (polynomiaux), en particulier le problème des ensembles disjoints. Ces transformations sont des cas particuliers de la réduction de Karp utilisée dans le cadre de la théorie de la NP-complétude [GJ79]. Comparer des problèmes polynomiaux est toujours très délicat. Par exemple, contrairement aux réductions de NP-complétude, des contraintes sur la tailles des données transformées sont nécessaires. Nous verrons que cet aspect est très restrictif.

Ces transformations montrent que l'existence d'un algorithme efficace (*i.e.* une complexité inférieure à $O(n^2)$ où n est le nombre de sommets du graphe) pour résoudre le problème du diamètre d'un graphe est peu probable. Ce résultat reste valide même si nous restreignons le graphe d'entrée à un graphe triangulé ou un graphe sans triplet astéroïde. Autrement dit les propriétés structurelles très fortes de ces graphes n'apportent rien au calcul du diamètre. L'algorithme classique semble encore le meilleur.

2.1 Heuristique et définitions

Commençons par rappeler quelques définitions. La *longueur* d'un chemin entre deux sommets est le nombre de ses arêtes.

Définition 2.1 *La* ***distance*** *entre deux sommets x et y, notée $d(x,y)$, est la* *longueur minimum d'un chemin allant de x à y.*

Définition 2.2 *L'**excentricité** d'un sommet x, notée $exc(x)$, est la distance maximum de x à tout autre sommet y.*

Définition 2.3 *Le **diamètre** d'un graphe, noté $diam(G)$, est le maximum des excentricités de ses sommets.*

Présentons l'heuristique. Elle ne retourne pas une distance, mais un sommet. Le résultat qui nous intéresse est l'excentricité de ce sommet. Un simple parcours en largeur à partir de ce sommet permet de la calculer.

Heuristique 7: Double Lex-BFS

Données : Un graphe G
Résultat : Un sommet v
début
  Soit w un sommet quelconque;
  $u \leftarrow$ le dernier sommet numéroté par Lex-BFS(w);
  $v \leftarrow$ le dernier sommet numéroté par Lex-BFS(u);
  retourner v;
fin

Soit G le graphe d'entrée de l'heuristique 7. Nous allons montrer que l'excentricité du sommet retourné est égale à $diam(G)$ ou $diam(G)-1$ si G est un graphe triangulé ou un graphe sans triplet astéroïde.

Définition 2.4 *Un **triplet astéroïde** est un triplet de sommets (x, y, z) tel qu'il existe un chemin entre tout couple de sommets parmi $\{x, y, z\}$ qui évite le voisinage du troisième.*

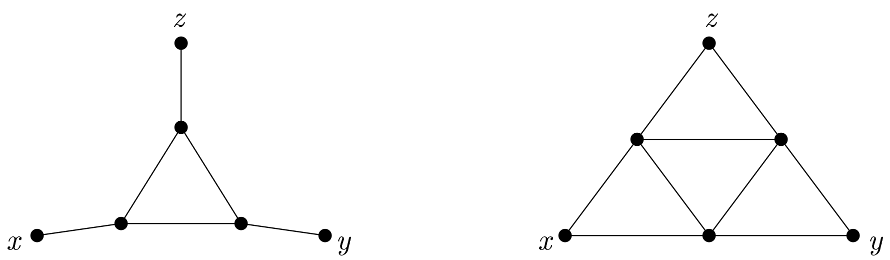


FIG. 2.1 – *Deux exemples de triplets astéroïdes (x, y, z).*

La notion de triplet astéroïde a été introduite par Lekerkerker et Boland [LB92] pour caractériser les graphes d'intervalles. Ils ont montré qu'un graphe est un graphe d'intervalles si et seulement si c'est un graphe triangulé qui ne contient pas de triplet astéroïde. La classe des graphes sans triplet astéroïde contient donc les graphes d'intervalles. Les triplets astéroïdes ont aussi été étudiés par Gallai [Gal67] dans le cadre de l'orientation transitive d'un graphe.

Corneil, Olariu et Stewart [COS97a, COS95] ont étudié les graphes sans triplet astéroïde. Ils ont mis en évidence l'existence d'une *paire dominante* pour ces graphes et ont prouvé que les derniers sommets obtenus par deux parcours Lex-BFS successifs sont une paire dominante.

Définition 2.5 *Une **paire dominante** est une paire de sommet (x, y) telle que pour tout chemin P_{xy} entre x et x et tout sommet z du graphe, soit z appartient à Pxy, soit z à un voisin sur P_{xy}.*

Nous pouvons dès maintenant remarquer que si (x, y) est une paire dominante, alors $diam(G) - 2 \leqslant d(x, y) \leqslant d(G)$.

De plus, l'existence d'une paire dominante (x, y) dans un graphe sans triplet astéroïde montre qu'étant donné un chemin P_{xy} entre x et y, le graphe s'organise dans un "cylindre" de rayon 1 autour de P_{xy}. Les graphes sans triplet astéroïde ont donc une structure linéaire. De ce point de vue, il est raisonnable d'espérer de bons résultats pour le calcul du diamètre sur cette classe de graphes.

Nous avons vu au chapitre 1 que Lex-BFS avait aussi de fortes propriétés sur les graphes triangulés. Ces graphes n'ont pas une structure linéaire comme les graphes sans triplet astéroïde, mais une structure arborée très forte (existence d'un arbre des cliques). De plus, ils possèdent aussi des propriétés de convexité intéressantes (voir [FJ86]). Ces raisons nous autorisent aussi à espérer quelques résultats pour le diamètre sur les graphes triangulés.

2.2 Résultats

En découvrant l'heuristique 7, la question naturelle porte sans doute sur la raison de l'utilisation de Lex-BFS. Un premier élément de réponse est fourni par le lemme ci-dessous.

Lemme 2.6 *Soit G un graphe triangulé ou un graphe sans triplet astéroïde. Soit u le dernier sommet numéroté par Lex-BFS. Alors il existe un sommet x d'excentricité maximum tel que $d(u, x) = exc(u)$.*

Cela signifie qu'un second parcours en largeur à partir du sommet u a des chances de terminer un sommet d'excentricité maximum. Cependant le lemme 2.8 va nous permettre de montrer que l'heuristique 7 peut échouer (*i.e.* ne pas trouver un sommet d'excentricité maximum). Notons qu'il exprime la principale différence entre Lex-BFS et un parcours en largeur classique.

Notation 2.7 *Nous définissons la distance entre un sommet x et un ensemble de sommets S, notée $d(x, S)$, comme la distance minimum entre x et un sommet de S.*

Lemme 2.8 *Soit S l'ensemble des sommets numérotés par Lex-BFS. Soient $x \notin S$ et $y \notin S$ tels que $d(x, S) < d(y, S)$, alors x sera numéroté avant y.*

Preuve : Faisons une preuve par récurrence. Supposons que x ait un voisin z dans S et que y n'en ait pas : $d(x, S) = 1$ et $d(y, S) > 1$. La marque de x (qui contient z) est donc plus grande que la marque de y (qui est vide). Or lorsqu'un sommet est numéroté, il est rajouté en fin de marque de ses voisins non numérotés. L'ordre lexicographique implique donc que si, à une étape donnée de Lex-BFS, la marque de x est plus grande que la marque de y, alors elle le restera toujours. Et donc x sera numéroté avant y.

Supposons que la propriété soit vérifiée pour sommet t tel que $d(t, S) = k$ et que $d(x, S) = k + 1$. Les prochains sommets numérotés seront tous les sommets ayant un voisin dans S. Et en particulier les sommets ayant un voisin dans S et appartenant à un plus court chemin entre x et S ou un plus court chemin entre y et S. Lorsque ces sommets seront numérotés, nous aurons $d(x, S') = k$ et $d(y, S') > d(x, S')$. Donc x sera numéroté avant y. □

Remarque 2.9 *L'heuristique 7 échoue sur les exemples des figures 2.2 et 2.3. Sur ces exemples, il est possible qu'une infinité de parcours Lex-BFS successifs partant sur le dernier sommet visité par le parcours précédant ne termine jamais un sommet d'excentricité maximum.*

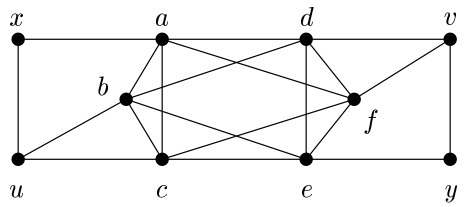


FIG. 2.2 – *G est un graphe sans triplet astéroïde, pour lequel une infinité de parcours Lex-BFS peut ne pas trouver un sommet d'excentricité maximum.*

Sur le graphe de la figure 2.2, les sommets d'excentricité maximum sont x et y. Or un parcours de Lex-BFS commençant en u finira toujours en v. En effet, Lex-BFS visite d'abord les voisins de u, *i.e.* x, b, c. Quelque soit l'ordre dans lequel il les visite, les sommets suivants seront a puis e. A ce stade l'ensemble des sommets numérotés est $S = \{u, x, a, b, c, e\}$. Puisque y est voisin de e et que v n'a aucun voisin numéroté dans S, y sera toujours visité avant v. Comme le graphe est symétrique, tout parcours Lex-BFS commençant en v finit aussi en u. Donc une infinité de parcours Lex-BFS ne terminera jamais x ou y.

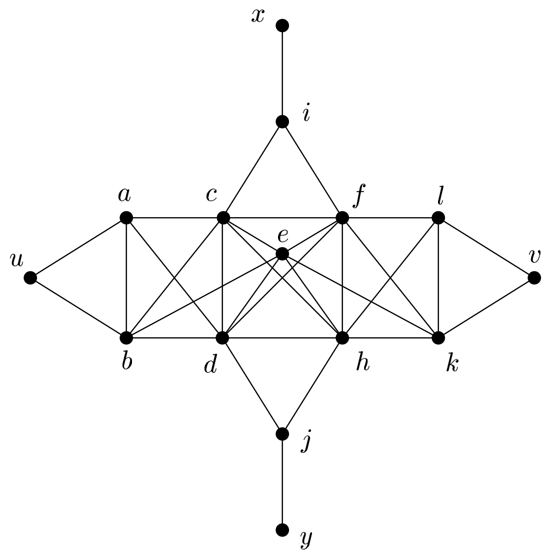


FIG. 2.3 – *G est un graphe triangulé pour lequel une infinité de parcours Lex-BFS peut ne pas trouver un sommet d'excentricité maximum*

Sur le graphe de la figure 2.3, les sommets d'excentricité maximum sont x et y. Un parcours de Lex-BFS commençant en u numérotera d'abord a et b. Les sommets c et d sont les seuls voisins communs à a et b, ils seront donc les sommets suivants dans l'ordre Lex-BFS. A ce stade, l'ensemble des sommets numérotés est $S = \{u, a, b, c, d\}$. Or $d(x, S) = d(y, S) = 2$ et $d(v, S) = 3$ et tous les autres sommets vérifient $d(z, S) \leqslant 2$. Donc v sera le dernier sommet numéroté. A nouveau, puisque le graphe est symétrique, tout Lex-BFS commençant en v finira en u. Donc une infinité de parcours Lex-BFS ne terminera jamais x ou y.

Théorème 2.10 *Soit G un graphe triangulé ou un graphe sans triplet astéroïde. L'heuristique 7 trouve un sommet v tel que $diam(G) - 1 \leqslant exc(v) \leqslant diam(G)$.*

Il s'agit donc après une exécution de l'heuristique 7 de déterminer dans quel cas nous sommes. Les deux prochains théorèmes mettent en évidence l'existence de sous-graphes induits lorsque l'algorithme ne trouve pas un sommet d'excentricité maximum.

Théorème 2.11 *Soit G un graphe triangulé. Si l'heuristique 7 ne trouve pas un sommet d'excentricité maximum, alors il existe un 3-soleil ou un 4-soleil.*

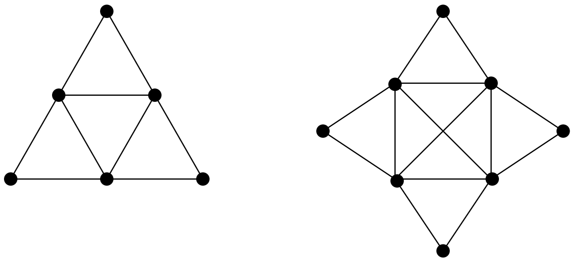


FIG. 2.4 – *Le 3-soleil et le 4-soleil.*

Théorème 2.12 *Soit G un graphe sans triplet astéroïde tel que $diam(G) = k$. Si l'heuristique 7 ne trouve pas un sommet d'excentricité maximum, alors il existe une $k-1$-échelle ou une $k-1$-échelle dégénérée.*

FIG. 2.5 – *La k-échelle et la k-échelle dégénérée.*

Il est évident que les théorèmes 2.11 et 2.12 ne sont pas des équivalences. Si l'heuristique 7 débute le premier parcours Lex-BFS par un sommet d'excentricité maximum, le dernier sommet du second parcours sera aussi un sommet d'excentricité maximum. D'autre part, en "jouant" un peu sur des exemples, nous remarquons vite que peu d'exécutions de l'heuristique 7 échouent. Pour affiner notre appréciation de l'heuristique, il serait intéressant de faire une étude probabiliste sur ce point.

Par ailleurs, il faut noter la dualité entre les sous-graphes détectés et les classes de graphes. Pour les graphes triangulés, le 3-soleil et le 4-soleil contiennent des triplets astéroïdes. Pour les graphes sans triplet astéroïde, la $k-1$-échelle et la

$k-1$-échelle dégénérée contiennent de nombreux cycles à 4. Ainsi dans les graphes pour lesquels l'heuristique échoue n'appartiennent pas à l'intersection des deux classes.

Par ailleurs, nous devons remarquer la conséquence directe du théorème 2.11 pour la classe des *graphes fortements triangulés*. Pour caractériser cette classe de graphes, nous devons généraliser les 3 et 4-soleil au k-soleil. Un *k-soleil* est un graphe composé d'un stable $\mathcal{S}_k = \{s_1, \ldots, s_k\}$ et d'une clique $\mathcal{C}_k = \{c_1, \ldots, c_k\}$ tels que s_i est adjacent à c_i et $c_{(i+1) mod k}$.

Définition 2.13 *Un graphe G est **fortement triangulé** si et seulement si il ne contient pas de k-soleil, $k \geqslant 3$.*

Corollaire 2.14 *L'heuristique 7 calcule la valeur exacte du diamètre d'un graphe fortement triangulé.*

Terminons sur un résultat amusant pour les graphes triangulés :

Théorème 2.15 *Si $diam(G)$ est pair, alors l'heuristique 7 termine sur un sommet d'excentricité maximum.*

Corollaire 2.16 *Si l'heuristique 7 termine sur un sommet dont l'excentricité est impaire alors c'est un sommet d'excentricité maximum.*

2.3 Transformations polynomiales

L'objectif de cette section est de montrer finalement que pour les deux classes de graphes étudiées, il ne faut guère espérer de meilleurs résultats que ceux présentés dans la section précédentes. Tout au plus pourrions nous élargir les sous-classes sur lesquelles l'heuristique trouve le diamètre.

Pour cela, nous comparons des problèmes. Nous souhaitons répondre à des questions du type : Le problème $\mathcal{A}$ est-il plus "facile" que le problème $\mathcal{B}$? Le problème $\mathcal{A}$ est-il "équivalent" au problème $\mathcal{B}$? Nous devons avant de comparer des problèmes préciser ces notions de "problème plus facile" et "problème équivalent".

Ne connaissant pas d'algorithme efficace pour le calcul du diamètre d'un graphe dans le cas général, certains auteurs [BKM95] ont cherché à se limiter au sous-problème suivant. Nous le comparons au produit de matrices booléennes, mais surtout au problème des ensembles disjoints.

Problème : $diam(G) > 2$

Données : *Un graphe $G = (V, E)$*
Question : *Tester si le diamètre de G est strictement supérieur à 2, $diam(G) > 2$*

2.3.1 Rappels et définitions

Rappelons dans un premier temps les notations O, Ω, Θ.

- $O(f) = \{g : \mathbb{N} \to \mathbb{N} | \exists c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, g(n) \leqslant cf(n)\}$
- $\Omega(f) = \{g : \mathbb{N} \to \mathbb{N} | \exists c > 0, \exists n_0 \in \mathbb{N} \text{ tels que } \forall n \geqslant n_0, g(n) \geqslant cf(n)\}$
- $\Theta(f) = O(f) \cap \Omega(f)$

Nous nous inspirons d'un protocole proposé par Shamos et Preparata [PS85]. Un problème $\mathcal{A}$ est transformé en un problème $\mathcal{B}$ de la manière suivante :

1. Transformer la donnée du problème $\mathcal{A}$ en une donnée pour le problème $\mathcal{B}$.
2. Résoudre le problème $\mathcal{B}$ avec la donnée transformée.
3. Transformer la solution du problème $\mathcal{B}$ en une solution du problème $\mathcal{A}$.

Si ces trois opérations peuvent être faites, nous dirons que $\mathcal{A}$ est *transformable* en $\mathcal{B}$. Il faut cependant faire attention. Puisque nous nous intéressons à des problèmes polynomiaux, nous ne pouvons pas nous contenter de transformer les données de $\mathcal{A}$ en des données de $\mathcal{B}$ qui sont de tailles polynomiale en la taille de celles de $\mathcal{A}$. Définissons plus formellement la $\tau(n)$*-transformation* où $\tau(n)$ est un polynôme en n :

Définition 2.17 *Soient $\mathcal{A}$ et $\mathcal{B}$ deux problèmes, nous dirons que $\mathcal{A} \preccurlyeq_{\tau(n)} \mathcal{B}$ si toute instance de taille n du problème $\mathcal{A}$ peut être résolue en exécutant un nombre constant de fois un algorithme pour le problème $\mathcal{B}$ sur des instances de taille linéaire en n et en utilisant une complexité additionnelle en $\Theta(\tau(n))$.*

La complexité additionnelle correspond au coût de la transformation d'une instance de taille n du problème $\mathcal{A}$ en une instance du problème B de taille linéaire en n. Nous pouvons donc aussi définir la notion d'équivalence entre problèmes :

Définition 2.18 *Les problèmes $\mathcal{A}$ et $\mathcal{B}$ seront donc dits* $\tau(n)$-équivalents *si et seulement si $\mathcal{A} \preccurlyeq_{\tau(n)} \mathcal{B}$ et $\mathcal{B} \preccurlyeq_{\tau(n)} \mathcal{A}$. Nous le noterons $\mathcal{A} \approx_{\tau(n)} \mathcal{B}$.*

La figure 2.6 illustre les deux lemmes suivants. Elle indique comment les relations définies peuvent être utilisées. A la différence de Preparata et Shamos, pour manipuler plus simplement les relations sur les bornes inférieures, nous imposons la notation Θ dans la complexité additionnelle. Nous verrons que ce n'est pas contraignant car les transformations sont souvent triviales.

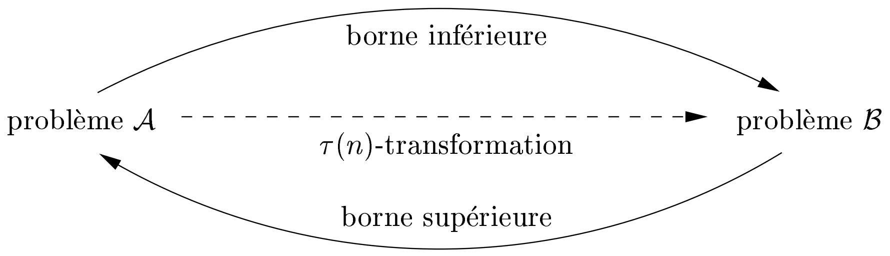


FIG. 2.6 – *Bornes inférieures et supérieures par la $\tau(n)$-transformation*

Lemme 2.19 (Borne supérieure) *[PS85] Si un problème $\mathcal{B}$ peut être résolu avec une complexité $T(n)$ et que $\mathcal{A} \preccurlyeq_{\tau(n)} \mathcal{B}$, alors le problème $\mathcal{A}$ peut être résolu avec une complexité au plus $T(n) + \Theta(\tau(n))$.*

Lemme 2.20 (Borne inférieure) *[PS85] Si un problème $\mathcal{A}$ demande une complexité supérieure ou égale à $T(n)$ et que $\mathcal{A} \preccurlyeq_{\tau(n)} \mathcal{B}$, alors le problème $\mathcal{B}$ nécessite en complexité supérieure ou égale à $T(n) - \Theta(\tau(n))$.*

2.3.2 Produit de matrices booléennes

Comparons le problème précédent au produit de matrices booléennes. Supposons que les graphes manipulés aient beaucoup d'arêtes, *i.e.* de l'ordre de $\Theta(n^2)$. Cette hypothèse peut sembler restrictive. Rappelons que nous souhaitons montrer la difficulté d'un problème $\mathcal{A}$ par rapport à un problème $\mathcal{B}$. Donc si nous pouvons transformer une infinité d'instances du problème $\mathcal{A}$ en instances du problème $\mathcal{B}$, ce sera suffisant pour notre propos.

Lemme 2.21 *Soit une classe $\mathcal{C}$ de graphes telle que $m \in \Theta(n^2)$ et $G \in \mathcal{C}$, alors :*

$$\text{“}diam(G) > 2\text{” pour } G \in \mathcal{C} \preccurlyeq_{n^2} \text{“produit de matrices booléennes”}$$

Preuve : Soit $G = (V, E)$ un graphe appartenant à $\mathcal{C}$. Obtenir une représentation en matrice d'adjacence du graphe G se fait en $\Theta(n^2)$. Soit $M(G)$ cette matrice.

Le graphe G^2 est un graphe ayant les mêmes sommets que G et tel que deux sommets x et y sont adjacents si et seulement si $d(x, y) \leqslant 2$ dans G. Il est clair que G^2 est complet si et seulement si $diam(G) \leqslant 2$. Or la matrice de G^2 est exactement la matrice $M(G)^2$ et $M(G)^2$ est la matrice booléenne complète (ne contenant que des 1) si et seulement si $diam(G) \leqslant 2$. Tester si une matrice $n \times n$ est complète peut se faire en $O(n^2)$. Finalement, si $\textsc{mm}(n)$ est le coût du produit de matrices booléennes $n \times n$, étant donné la matrice d'adjacence d'un graphe G, nous pouvons tester en $O(\textsc{mm}(n))$ si $diam(G) > 2$. □

Si nous ne faisons aucune hypothèse sur le nombre d'arêtes, il est raisonnable de penser qu'un algorithme efficace pour le produit de matrices utilise des matrices creuses. Dans ce cas, l'espace utilisé peut être considéré de l'ordre de $O(n+m)$ et le résultat ci-dessus reste alors valide.

Enfin si nous souhaitons tester si le diamètre du graphe est k, une constante fixée, il suffit de calculer la puissance k de la matrice d'adjacence. Nous obtenons donc un coût du même ordre.

La transformation présentée permet juste de dire que s'il existe un algorithme efficace pour le produit de matrices booléennes, alors le problème $diam(G) > 2$ peut se résoudre dans la même efficacité. Actuellement, le produit de matrices booléennes peut se faire en $O(n^2 \log n)$. Même si nous n'avons pas montré la transformation réciproque, nous ne croyons pas à l'existence d'un algorithme de complexité inférieure à $O(n^2)$ pour le calcul du diamètre d'un graphe. Pour cela, nous allons considérer un second problème.

2.3.3 Problème des ensembles disjoints

Comparons le problème du diamètre à celui des "ensembles disjoints" :

Problème : *Ensembles disjoints*

Données : *Une famille $\mathcal{F}$ de sous-ensembles d'un ensemble de base X*
Question : *Existe-t-il S et S' appartenant à $\mathcal{F}$ tels que $S \cap S' = \emptyset$?*

La taille d'une famille $\mathcal{F}$ de sous-ensembles de X est : $|\mathcal{F}| + |X| + \sum_{S \in \mathcal{F}} |S|$, la taille du biparti d'incidence $(X, \mathcal{F})$.

Lemme 2.22 *Soit G un graphe quelconque :*

$$\text{“}diam(G) > 2\text{”} \preccurlyeq_{n+m} \text{“ensembles disjoints”}$$

Preuve : Soit $G = (V, E)$ un graphe. Le voisinage clos d'un sommet x, noté $N[x]$, est $N(x) \cup \{x\}$. Considérons la famille d'ensembles $\mathcal{F}_G$ constituée des voisinages clos des sommets de V : $\mathcal{F}_G = \{N[x] : x \in V\}$. Clairement la taille de la famille $\mathcal{F}_G$ est linéaire en la taille de G. Elle peut être construite en $\Theta(n+m)$.

S'il existe deux sommets x et y tels que $N[x] \cap N[y] = \emptyset$, alors $diam(G) > 2$. Dans le cas contraire, pour tout x, y de sommets, soit x et y sont adjacents, soit ils ont un voisin commun. Donc $diam(G) > 2$ si et seulement si il existe deux ensembles disjoints dans la famille $\mathcal{F}_G$. □

Soit $\mathcal{F} = \{S_i \subseteq X : 1 \leqslant i \leqslant k\}$ une famille d'ensembles. Posons $n = |X|$ et $m = \sum_i |S_i|$. Pour montrer l'autre direction, nous devons à nouveau faire l'hypothèse que $m \in \Theta(n^2)$. Nous avons alors :

Lemme 2.23 *Soit $\mathcal{H}$ une classe de familles de sous-ensembles de X telle que $m \in \Theta(n^2)$ et $\mathcal{F} \in \mathcal{H}$, alors :*

"ensembles disjoints" pour la classe $\mathcal{H} \preccurlyeq_{n^2}$ "$diam(G) > 2$"

Preuve : La famille $\mathcal{F}$ peut être transformée en un graphe $G = (V, E)$ dont les sommets sont les éléments de X et les éléments S_i de $\mathcal{F}$. Le sous-graphe induit par les éléments de X est une clique, celui induit par les éléments de $\mathcal{F}$ est un stable. Il existe une arête entre un sommet x correspondant à un élément de X et un sommet s correspondant à un élément S de $\mathcal{F}$ si et seulement si x est un élément de S. Clairement ce graphe à une taille en $O(n^2 + m)$. Comme $m = O(n^2)$, il est linéaire en la taille de $\mathcal{F}$. Il peut être construit en $\Theta(n^2)$.

S'il existe deux ensembles disjoints S_i et S_j dans $\mathcal{F}$, alors la distance dans G entre les sommets correspondants s_i et s_j est 3. Ces sommets ne sont pas adjacents et n'ont pas de voisins communs. Par contre, tous leurs voisins forment une clique. Dans le cas contraire (il n'existe pas d'ensembles disjoints), tout couples de sommets s_i, s_j possède un voisin commun. La distance les séparant est donc 2. De plus la distance entre deux sommets x, x' correspondant à des éléments de X est 1. Enfin la distance entre un sommet x et un sommet s_i est au plus 2. Donc il existe deux ensembles disjoints dans $\mathcal{F}$ si et seulement si $diam(G) > 2$. □

Nous obtenons donc finalement :

Théorème 2.24 *Soient $\mathcal{H}$ une classe de familles de sous-ensembles de X telles que $m \in \Theta(n^2)$ et $\mathcal{F} \in \mathcal{H}$, alors :*

"ensembles disjoints" pour la classe $\mathcal{H} \approx_{n^2}$ "$diam(G) > 2$"

Remarque 2.25 *Evoquons une transformation due à Preparata et Shamos [PS85] : soient deux ensembles d'entiers S_1 et S_2 de taille n. Le problème consistant à déterminer si S_1 et S_2 ont au moins un élément en commun, peut être transformé en calcul du diamètre d'un ensemble de points dans le plan. Certes le calcul du diamètre d'un graphe et du diamètre d'un ensemble de points dans le plan ne sont pas les mêmes, mais cette relation nous semble intéressante à mentionner.*

Il suffit, comme le montre la figure 2.7, de placer chaque point k de S_1 sur l'intersection du quart inférieur gauche d'un cercle de diamètre 1 et de la droite $y = kx$. Les points de S_2 sont placés de la même manière sur le quart supérieur *droit de ce cercle. Alors S_1 et S_2 ont un voisin commun si et seulement si le diamètre de l'ensemble des points ainsi placés est 1.*

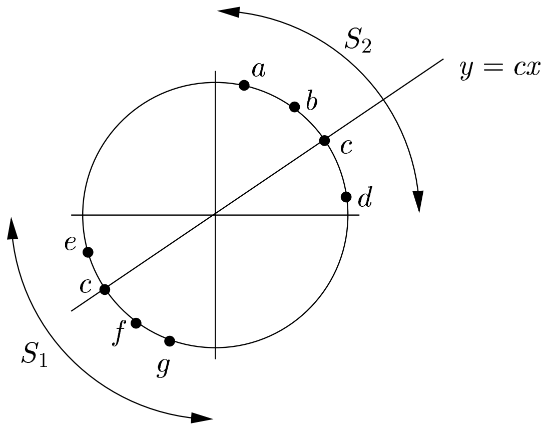


FIG. 2.7 – $S_1 = \{c, e, f, g\}$ *et* $S_2 = \{a, b, c, d\}$

Reingold [Rei72] a montré que tester si S_1 et S_2 ont une intersection non-vide demande $\Omega(n \log n)$ comparaisons.

2.3.4 Graphes triangulés

Dans cette section, nous comparons le calcul du diamètre des graphes triangulés au problème des ensembles disjoints. Pour montrer une quelconque relation entre ces deux problèmes, il suffit de la montrer pour un ensemble infini d'instances. Nous pouvons donc nous restreindre à certaines sous-classes des graphes triangulés. Nous considérons les *(clique-stable)-graphe*s (split graphes en anglais).

Définition 2.26 *Un graphe $G = (V, E)$ est un* ***(clique-stable)-graphe*** *si et seulement si V peut être partitionné en deux ensembles $\mathcal{S}$ et $\mathcal{C}$ tels que $\mathcal{S}$ induit un stable et $\mathcal{C}$ une clique.*

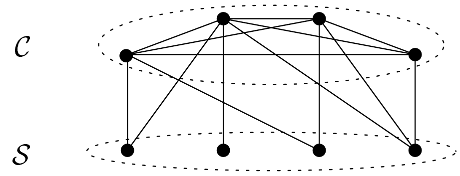


FIG. 2.8 – *Un (clique-stable)-graphe*

Notons que toute arête entre un sommet de $\mathcal{S}$ et un sommet de $\mathcal{C}$ est possible. Dans la suite, nous supposerons qu'un (clique-stable)-graphe n'est ni une clique, ni un stable.

Problème : *Diamètre d'un (clique-stable)-graphe*

Données : *Un (clique-stable)-graphe* $G = (V, E)$
Question : *Déterminer le diamètre de* G

Lemme 2.27 *Soit* $G = (V, E)$ *un (clique-stable)-graphe connexe. Si* G *n'est pas une clique, alors* $2 \leqslant diam(G) \leqslant 3$.

Preuve : Puisque G n'est pas une clique, il existe deux sommets x et y non-adjacents. Donc x ou y n'appartient pas à la clique $\mathcal{C}$. Si $x \in \mathcal{C}$ et $y \in S$ alors ils possèdent un voisin commun (G est connexe). Donc $d(x, y) = 2$. Supposons que x et y soient des sommets de S. Si x et y possèdent un voisin commun alors $d(x, y) = 2$ sinon $d(x, y) = 3$. Donc $diam(G) = 3$ si et seulement si il existe deux sommets x et y de S tels que $N(x) \cap N(y) = \emptyset$. Dans le cas contraire $diam(G) = 2$. □

Le lemme 2.22 prouve directement le lemme suivant. Cependant nous proposons une transformation légèrement différente et un peu plus simple.

Lemme 2.28 *"diamètre d'un (clique-stable)-graphe"* $\preccurlyeq_{n+m}$ *"ensembles disjoints".*

Preuve : Pour transformer un (clique-stable)-graphe $G = (V, E)$ en une instance du problème des ensembles disjoints, il suffit de considérer la famille $\mathcal{F}$ des voisinages des sommets du stable. Le lemme 2.27 a montré que $diam(G) = 3$ si et seulement si il existe deux sommets x et y tels que $N(x) \cap N(y) = \emptyset$. Autrement dit $diam(G) = 3$ si et seulement si il existe deux ensembles disjoints dans $\mathcal{F}$.

La reconnaissance des (clique-stable)-graphes se fait avec une complexité linéaire (voir par exemple [Gol80]). Sans augmenter la complexité, il est possible d'identifier le stable et la clique. La famille $\mathcal{F}$ peut donc être calculée en $\Theta(n + m)$ où $n = |V|$ et $m = |E|$. □

Pour l'autre direction, nous devons encore nous limiter à des classes $\mathcal{H}$ de familles de sous-ensembles de X telles que pour $\mathcal{F} \in \mathcal{H}$, nous avons $\sum_{S \in \mathcal{F}} |S| = m \in \Theta(n^2)$. Remarquons enfin que le graphe construit dans la preuve du lemme 2.23 est un (clique-stable)-graphe. Nous pouvons donc conclure que :

Théorème 2.29 *Soit* $\mathcal{H}$ *une classe de famille d'ensembles de* X *telles que pour* $\mathcal{F} \in \mathcal{H}$, *alors* $\sum_{S \in \mathcal{F}} |S| \in \Theta(|n|^2)$ *où* $n = |X|$.
"diamètre d'un (clique-stable)-graphe" $\approx_{n^2}$ *"ensembles disjoints" pour la classe* $\mathcal{H}$

Preuve : Nous devons juste faire attention qu'il y ait effectivement correspondance entre les (clique-stable)-graphes que nous considérons et les familles de la classe $\mathcal{H}$. Dans ce cas il suffit de considérer la classe $\mathcal{C}$ des (clique-stable)-graphes qui peuvent être construits à partir d'une famille $\mathcal{F} \in \mathcal{H}$. $\square$

2.3.5 Graphes sans triplet astéroïde

Intéressons-nous aux graphes sans triplet astéroïde. Comme dans la section précédente, nous considérons une sous-classe des graphes sans triplet astéroïde (définie pour la circonstance). Nous allons retrouver exactement le même type de résultats qu'à la section précédente.

Définition 2.30 *Un **graphe triclique sans triplet astéroïde** est un graphe $G = (V, E)$ sans triplet astéroïde tel que V peut être partitionné en trois cliques $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ où $\mathcal{C}_2$ est une copie de $\mathcal{C}_3$. Toute arête entre un sommet $x_1 \in \mathcal{C}_1$ et un sommet $x_2 \in \mathcal{C}_2$ est autorisée et x_1 a le même voisinage dans $\mathcal{C}_2$ que $\mathcal{C}_3$. (voir figure 2.9)*

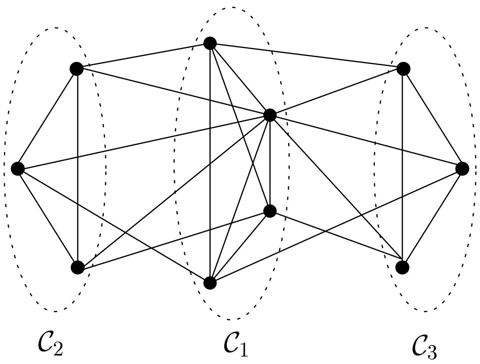


FIG. 2.9 – *Un graphe triclique sans triplet astéroïde*

Un graphe triclique ne peut contenir de triplet astéroïde. Pour choisir un triplet de sommets indépendants, il faut prendre un sommet par clique $x_1 \in \mathcal{C}_1$, $x_2 \in \mathcal{C}_2$ et $x_3 \in \mathcal{C}_3$. Or tout chemin de x_2 à x_3 passe par $\mathcal{C}_1$, *i.e.* intersecte le voisinage de x_1.

Pour la suite, nous supposons que les trois cliques $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ sont différentes et non-vides. Il faut noter que sous ces conditions, nous pouvons reconnaître ces graphes en temps linéaire grâce à deux parcours de Lex-BFS. L'idée est qu'en étudiant les étiquettes de Lex-BFS, nous pouvons déterminer les trois cliques. Puis avec un calcul de sommets jumeaux (voir chapitre 6), nous pouvons identifier, pour chaque sommet de $\mathcal{C}_2$, sa copie dans $\mathcal{C}_3$.

Problème : *Diamètre d'un graphe triclique sans triplet astéroïde*

Données : *Un graphe triclique sans triplet astéroïde* $G = (V, E)$
Question : *Déterminer le diamètre de* G

Lemme 2.31 *Si* $G = (V, E)$ *est un graphe triclique sans triplet astéroïde, alors* $2 \leqslant diam(G) \leqslant 3$.

Preuve : Soit x un sommet de C_1. La distance entre toute paire de sommets de C_1 est 1. La distance entre x et un sommet y de C_2 ou C_3 est au plus 2 dans le cas où ils ne sont pas voisins. Donc l'excentricité d'un sommet de C_1 est au plus 2.

Soit y un sommet de C_2. La distance entre y et tout autre sommet de C_2 est 1. La distance entre y est un sommet $x \in C_1$ est 1 s'ils sont voisins, 2 sinon. En revanche, la distance entre y et $z \in C_3$ est 2 si et seulement si y et z ont un voisin commun dans C_1. Dans le cas contraire, elle vaut 3. Ainsi l'excentricité d'un sommet de C_2 est soit 2, soit 3. De même pour les sommets de C_3.

L'excentricité maximum pour les sommets de G est toujours atteinte par un sommet de C_2 et un sommet C_3. Comme les trois cliques sont distinctes et non-vides, nous avons $2 \leqslant diam(G) \leqslant 3$. $\square$

Lemme 2.32 *"diamètre d'un graphe triclique sans triplet astéroïde "* $\preccurlyeq_{n+m}$ *"ensembles disjoints".*

Preuve : Pour transformer un graphe triclique sans triplet astéroïde $G = (V, E)$ en une instance du problème des ensembles disjoints, il suffit de considérer la famille $\mathcal{F}$ des voisinages des sommets de $\mathcal{C}_2$ et $\mathcal{C}_3$ dans la clique $\mathcal{C}_1$. Le lemme 2.31 a montré que $diam(G) = 3$ si et seulement si il existe $y \in C_2$ et $z \in C_3$ tels que $N_1(y) \cap N_1(z) = \emptyset$. Autrement dit $diam(G) = 3$ si et seulement si il existe deux ensembles disjoints dans la famille $\mathcal{F}$.

Nous avons indiqué comment les trois cliques $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ peuvent être calculées en linéaire. Dès lors, la famille $\mathcal{F}$ peut aussi être calculée en $\Theta(n + m)$ où $n = |V|$ et $m = |E|$. $\square$

Notons que le lemme 2.23 peut à nouveau être appliqué. Il faut toutefois modifier la construction du graphe à partir d'une famille d'ensembles $\mathcal{F}$ pour obtenir un graphe triclique sans triplet astéroïde. Pour cela, il suffit que le sous-graphe induit par les ensembles de $\mathcal{F}$ soit une clique et non un stable. Il reste alors à faire une copie de ce sous-graphe et des arêtes le connectant au sous-graphe induit par X. Nous obtenons donc :

Théorème 2.33 *Soit $\mathcal{H}$ une classe de familles d'ensembles de X telle que pour $\mathcal{F} \in \mathcal{H}$, alors $\sum_{S \in \mathcal{F}} |S| \in \Theta(|n|^2)$ où $n = |X|$.*
"diamètre d'un graphe triclique sans triplet astéroïde"
$\approx_{n^2}$ *"ensembles disjoints" pour la classe $\mathcal{H}$*

Preuve : Pour assurer la correspondance, nous pouvons considérer seulement la classe des graphes tricliques sans triplet astéroïde calculable à partir d'une famille $\mathcal{F} \in \mathcal{H}$. □

2.4 Conclusions

Un aspect intéressant de l'heuristique 7 proposée est la ressemblance des résultats pour les graphes triangulés et les graphes sans triplet astéroïde. Ces deux classes sont très différentes et cependant le même algorithme obtient des résultats très voisins. Pour calculer la valeur exacte du diamètre d'un graphe triangulé, il ne doit pas contenir de 3-soleil ou 4-soleil. Ces deux sous-graphes différencient les graphes triangulés des graphes sans triplet astéroïde. Pour les graphes sans triplet astéroïde, nous devons aussi interdire des sous-graphes (la k-échelle). Ces graphes contiennent des cycles de longueur 4, ils ne sont donc pas triangulés.

Nous sommes donc confrontés à une situation originale. En temps linéaire, nous savons calculer le diamètre d'une large part des graphes triangulés et des graphes sans triplet astéroïde. Mais pour tester si la valeur obtenue est la bonne, il faut lancer une procédure en $O(nm)$. Les transformations abordées dans la dernière section apparaissent de mauvais augure pour de meilleurs résultats du point de vue complexité. Même si nous n'avons pas montré l'équivalence avec le produit de matrices booléennes, le problème des ensembles disjoints montre la difficulté du calcul du diamètre d'un graphe.

Chapitre 3

Diamètre des graphes triangulés

Ce chapitre contient les preuves concernant les graphes triangulés des résultats énoncés au chapitre précédent. Celles sur les graphes sans triplet astéroïde sont dues au Pr. D. Corneil [Cor]. Nous ne les présenterons pas.

Puisque dans un arbre, il n'existe qu'un chemin entre deux sommets quelconques, les distances sont simples à évaluer. Les graphes triangulés peuvent être considérés comme une généralisation des arbres : un arbre est un graphe sans cycle, un graphe triangulé contient uniquement des cycles de longueur 3. De plus, l'existence d'un arbre de cliques (voir chapitre 1) laisse penser que les propriétés de distances sur les arbres doivent aussi pouvoir se généraliser aux graphes triangulés.

3.1 Parcours Lex-BFS sur les graphes triangulés

Les arbres de cliquesjouent un rôle central dans la preuve que nous proposons. Plus précisément, nous utilisons les arbres de cliques calculés par l'algorithme 5 (voir chapitre 1). Nous ferons donc quelques rappels sur Lex-BFS et le calcul d'un arbre de cliques. La définition et le lemme suivant donnent une idée de la manière dont les arbres de cliques interviennent dans nos preuves.

Définition 3.1 *Une* ***chaîne de cliques*** *est un ordonnancement d'un ensemble de cliques tel que pour chaque sommet x, les cliques contenant x sont consécutives.*

Lemme 3.2 *Soient x et y deux sommets distincts. Alors il existe une chaîne de cliques contenant tous les plus courts chemins entre x et y.*

Preuve : Il suffit de considérer un arbre de cliques $\mathcal{T}$. Soient C_x et C_y les plus proches cliques contenant respectivement x et y. Alors la chaîne de cliques entre

$[C_x = C_0, \ldots, C_l = C_y]$ entre C_x et C_y dans $\mathcal{T}$ contient tous les plus courts chemins entre x et y.

Supposons que $l = 1$. Par hypothèse $x \in C_x \setminus C_y$ et $y \in C_y \setminus C_x$, donc si (C_x, C_y) est une arête de $\mathcal{T}$, alors $C_x \cap C_y$ est un x, y-séparateur minimal. Dans ce cas, il existe des chemins de longueurs 2 entre x et y et ils intersectent tous $C_x \cap C_y$.

Supposons que $l > 1$. Puisque $\mathcal{T}$ est un arbre de cliques, pour tout $0 \leqslant j < l$, $S_{j,j+1} = C_j \cap C_{j+1}$ est un séparateur minimal pour $a_j \in C_j \setminus C_{j+1}$ et $A_{j+1} \in C_{j+1} \setminus C_j$. Puisque $x \in C_x \setminus C_1$ et $y \in C_y \setminus C_{l-1}$, $S_{j,j+1}$ est aussi un x, y-séparateur. Donc tout chemin entre x et y doit intersecter $S_{j,j+1}$ pour tout j tel que $0 \leqslant j < l$. Supposons qu'il existe un chemin μ entre x et y possédant un sommet z qui n'appartient à aucune clique C_k, $\leqslant k \leqslant l$. Alors z n'appartient à aucun séparateur $S_{j,j+1}$. Or tout sommet de $S_{j,j+1}$ est adjacent avec tout sommet de $S_{j+1,j+2}$. Puisque μ intersecte tout les séparateurs $S_{j,j+1}$, μ possèdent une corde : μ n'est pas un plus court chemin. $\square$

Dans le chapitre 1, nous avons proposé un algorithme de calcul d'un arbre de cliques à partir d'un ordre Lex-BFS σ d'un graphe triangulé G. Cet algorithme n'intervient pas dans l'algorithme de calcul du diamètre. En revanche, nous utilisons les propriétés des arbres qu'il construit. Rappelons donc brièvement comment un arbre de cliques est calculé à partir d'un ordre Lex-BFS (pour plus de précision se référer au chapitre 1).

Lex-BFS et les cliques maximales

Soit $\sigma = [x_n, \ldots, x_i, \ldots, x_1]$ un ordre Lex-BFS. Le sommet x_i est simplicial dans le graphe $G_i = G[x_n, \ldots, x_i]$. Soit $\mathcal{C}$ l'unique clique maximale de G_i contenant x_i. Si $\mathcal{C} \setminus \{x_i\}$ n'est pas une clique maximale de G_{i+1}, nous dirons que x_i *découvre* une nouvelle clique. Dire que les sommets d'une clique sont visités consécutivement signifie que, lorsqu'une nouvelle clique maximale $\mathcal{C}$ de G est découverte, alors tous les sommets non-numérotés de $\mathcal{C}$ sont numérotés consécutivement.

Arbre de cliques $\mathcal{T}$ associé à un ordre Lex-BFS

Rappelons la méthode utilisée au chapitre 1 pour construire un arbre de cliques. Lorsqu'une nouvelle clique est découverte par un sommet x dans σ, nous l'incluons dans l'arbre $\mathcal{T}$. Soient x' le dernier sommet qui a marqué x et $\mathcal{C}'$ la clique maximale découverte par x'. Nous connectons alors la clique $\mathcal{C}$ à la clique $\mathcal{C}'$ dans $\mathcal{T}$. Nous pouvons donc considérer l'arbre de cliques comme une arborescence.

Nous avions aussi noté que la marque de x est en fait l'intersection entre $\mathcal{C}$ et $\mathcal{C}'$ et qu'elle correspond à un séparateur minimal pour tout $x \in \mathcal{C} \setminus \mathcal{C}'$ et $x' \in$

$\mathcal{C}' \setminus \mathcal{C}$. Ceci signifie donc que Lex-BFS passe d'une clique maximale à une autre par l'intermédiaire des séparateurs minimaux. C'est ce que précise le lemme suivant :

Lemme 3.3 *Soient $G = (V, E)$ un graphe triangulé et S un séparateur minimal de G. Soient cc_1 et cc_2 deux composantes connexes de $G[V \setminus S]$ pour lesquelles S est minimal. Les sommets de S ne peuvent pas être numérotés par Lex-BFS après les sommets de cc_1 et de cc_2.*

Preuve : Pour montrer la validité de l'algorithme MIS, nous avons prouvé ce résultat (voir théorème 1.36). Puisque Lex-BFS est un cas particulier de MIS, le lemme est vérifié. □

Corollaire 3.4 *Soit P_{xy} un chemin sans corde dans le graphe triangulé G entre x et y. Alors à toute étape d'un parcours Lex-BFS, l'ensemble des sommets numérotés de P_{xy} est connexe sur P_{xy}.*

Preuve : Puisque P_{xy} est un chemin sans corde, tout sommet v interne de P_{xy} appartient à un t, t'-séparateur minimal pour t et t' se trouvant de part et d'autre de v. Il suffit alors d'appliquer le lemme précédent. □

Lex-BFS et les chemins sans corde

Corollaire 3.5 *Soit $\mathcal{S}_{xy}$ l'ensemble des sommets appartenant à un plus court chemin entre les sommets x et y du graphe triangulé G. Soit z le premier sommet numéroté de $\mathcal{S}_{xy}$ par σ, un ordre Lex-BFS. Si $d(x, z) = k$, alors les sommets $t \in \mathcal{S}_{xy}$ tels que $d(x, t) = k - i$ seront numérotés avant les sommets $t' \in \mathcal{S}_{xy}$ tels que $d(x, t) = k - (i + 1)$, pour $i > 0$.*

Preuve : Considérons le sous-graphe induit par les sommets de S_{xy}. Dans $G[S_{xy}]$, la suppression de l'ensemble D_k des sommets à distance k de x déconnecte x et y. Par contre, un seul de ces sommets permet de reconnecter x et y. Donc D_k est un séparateur minimal : $G[D_k]$ est une clique (voir théorème 1.17).

Puisque les sommets de D_k ne sont pas simpliciaux (dans G) et qu'ils forment une clique, D_k est contenu dans un x, y-séparateur minimal de G. De même les sommets de D_{k+1} sont contenus dans un autre x, y-séparateur minimal.

Dans ce cas le lemme 3.3, montre que les sommets de $\mathcal{S}_{xy}$ à distance $k - i$ de x sont numérotés avant ceux à distance $k - (i + 1)$. □

Notation 3.6 *Soit S l'ensemble des sommets numérotés à une étape donnée de Lex-BFS. Alors $N_s(x)$ désignera l'ensemble des sommets $z \in S$ tels que $d(x, z) = d(S, z)$.*

Enfin, dans le chapitre 2 nous avons montré que si x et y sont deux sommets tels que, à une étape donnée de Lex-BFS, la distance entre x et l'ensemble des sommets numérotés S est plus grande que la distance entre y et S, alors x sera numéroté après y. Nous montrons maintenant que, sous certaines conditions en cas d'égalité entre $d(S,x)$ et $d(S,y)$, nous pouvons savoir qui de x ou y sera numéroté le premier.

Lemme 3.7 *Soient G un graphe triangulé et σ un ordre Lex-BFS. Soit S l'ensemble des sommets numérotés à une étape donné. Si $x \in S$ et $y \in S$ ne sont pas adjacents, alors il n'existe pas entre x et y de chemin $P_{xy} = [x = v_0, v_1 \ldots v_k, y = v_{k+1}]$ tel que $\forall j,\ 1 \leqslant j \leqslant k,\ v_j \notin S$.*

Preuve : Supposons qu'il existe un chemin entre x et y n'utilisant que des sommets extérieurs à S. Soient $P_{xy} = [x = v_0, v_1 \ldots v_k, y = v_{k+1}]$ le plus court et v_j, $1 \leqslant j \leqslant k$, le dernier sommet de ce chemin numéroté par σ. Puisque x et y ne sont pas adjacents, $j > 0$. Rappelons que x et y sont les premiers sommets numérotés de P_{xy}. Les sommets v_{j-1} et v_{j+1} ne sont pas adjacents, en effet P_{xy} est un plus court chemin. Or ils sont numérotés avant v_j. Donc v_j n'est pas simplicial. □

Corollaire 3.8 *Soient $P = [x_1, \ldots, x_p]$ un chemin sans corde d'un graphe triangulé G et σ un ordre Lex-BFS. Soit $[x_k, \ldots, x_h]$ le sous-chemin des sommets numérotés de P à une étape donnée de σ. Alors avant de numéroter x_{k-1} (resp. x_{h+1}), σ doit numéroter tous les voisins dans G de x_{k+1} (resp. x_{h-1}), et en particulier, l'ensemble des cliques maximales contenant l'arête (x_{k+1}, x_k) (resp. (x_{h-1}, x_h)).*

Preuve : Prouvons le résultat pour les voisins de x_{k+1}. La même démonstration est valable pour x_{h-1}. Soit a un voisin de x_{k+1}. Puisque P est un chemin sans corde, aucun sommet b de P entre x_1 et x_k n'est voisin de x_{k+1}. Le lemme 2.8 montre que a sera numéroté avant b. □

Lemme 3.9 (lemme d'inclusion) *Soient G un graphe triangulé et σ un ordre Lex-BFS. Soient x et y deux sommets tels que $d(x,S) = d(y,S)$. Si $N_s(x) \subsetneq N_s(y)$ alors y sera numéroté avant x.*

Preuve : Soit $d(x,S) = d(y,S) = k$. Cette propriété est vraie par définition de l'ordre lexicographique si $k = 1$. Supposons que $k > 1$.

Soient $y' \in N_s(y) \setminus N_s(x)$ et $x' \in N_s(y) \cap N_s(x)$. D'après le lemme 3.7, $N_s(y)$ est une clique, donc x' et y' sont adjacents. Considérons des plus courts chemins $P_{yy'} = [y = a_k, \ldots, a_1, a_0 = y']$ et $P_{yx'} = [y = b_k, \ldots, b_1, b_0 = x']$. Alors les sommets de $P_{yy'} \cup P_{yx'}$ induisent un cycle. Puisque $P_{yx'}$ et $P_{yx'}$ sont deux plus courts chemins, les seuls voisins possibles pour x' et y' sont a_1 et b_1. Pour trianguler le cycle, au moins une des 2 arêtes (y', b_1), (x', a_1) doit exister. Supposons que (y', b_1) existe. Alors $[y, \ldots, a_1, x', b_1, \ldots, y]$ est un cycle de longueur supérieure ou égale à quatre. Dans ce cycle, x' ne peut pas avoir d'autres voisins que a_1 et b_1. Donc l'arête (a_1, b_1) doit exister.

Notons $N'_s(y)$ l'ensemble des sommets c tels que c ayant un voisin dans $N_s(y)$ et appartenant à un plus court chemin entre y et $N_s(y)$. Nous venons de montrer que $N'_s(y)$ est une clique.

Montrons qu'il existe un sommet $\alpha \in N'_s(y)$ tel que $N_s(c) = N_s(y)$. Supposons le contraire. Soient α_1 et α_2 deux sommets tels que $N_s(\alpha_1)$ et $N_s(\alpha_2)$ sont maximaux pour l'inclusion (α_1 et α_2 sont adjacents). Il existe donc $y_1 \in N_s(\alpha_1) \setminus N_s(\alpha_2)$ et $y_2 \in N_s(\alpha_2) \setminus N_s(\alpha_1)$ (y_1 et y_2 sont adjacents). Alors le cycle $\alpha_1, y_1, y_2, \alpha_2$ est un cycle sans corde : impossible, donc le sommet α tel que $N_s(\alpha) = N_s(y)$ existe.

Autrement dit α est étiqueté par $N_x(y)$. De même, nous pouvons montrer l'existence d'un sommet $\delta \in N'_s(x)$ étiqueté par $N_s(x)$. Dans $N'_s(y)$ (*resp.* $N'_s(x)$), α (*resp.* δ) est le premier sommet numéroté. Or $N_s(x) \subsetneq N_s(y)$, donc α sera numéroté avant δ. Soit S l'ensemble des sommets numérotés lorsque α est numéroté mais pas δ. Nous avons alors $d(y, S) = k - 1$. Notons que $d(\alpha, x) > k$, sinon $N_s(x) = N_s(y)$. Donc nous avons toujours $d(x, S) = k$. Le lemme 2.8 montre alors que y est numéroté avant x. □

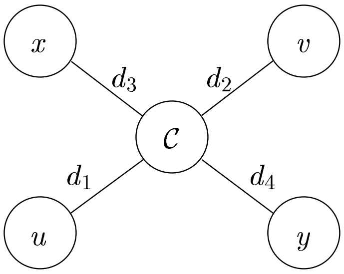
3.2 Résultats principaux

Nous allons montrer maintenant que, si u est le dernier sommet numéroté par un ordre Lex-BFS σ_1, alors il existe, dans les sommets à distance maximum de u, un sommet d'excentricité maximum. Une telle propriété est absolument nécessaire, si l'on espère terminer par un sommet d'excentricité maximum à la fin du second Lex-BFS.

Théorème 3.10 *Soient G un graphe triangulé et u le dernier sommet numéroté par un parcours Lex-BFS σ_1. Alors il existe un sommet simplicial z d'excentricité maximum tel que $exc(u) = d(u, z)$.*

Preuve : Supposons qu'un tel sommet z n'existe pas. Soit v le dernier sommet numéroté par un parcours Lex-BFS σ_2 commençant en u. Ainsi v est un sommet à distance maximum de u et son excentricité n'est pas maximum. Soit $\mathcal{T}$ l'arbre de cliques associé à σ_2. Notons x et y deux sommets d'excentricité maximum. Nous allons envisager les différentes configurations possibles entre la u, v-chaîne de cliques $\mathcal{P}_{uv}$ et la x, y-chaîne de cliques $\mathcal{P}_{xy}$ dans $\mathcal{T}$. Il y en a trois possibles : elles ont une intersection vide, elles s'intersectent sur une seule clique maximale ou elles s'intersectent sur une chaîne de cliques.

I. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est une clique $\mathcal{C}$:



Définissons quelques notations : $d_1 = d(u, \mathcal{C})$, $d_2 = d(v, \mathcal{C})$, $d_3 = d(x, \mathcal{C})$ et $d_4 = d(y, \mathcal{C})$. La distance entre u et v est donc : $d(u, v) = d_1 + \varepsilon_{12} + d_2$ où ε_{12} désigne le nombre d'arêtes de $\mathcal{C}$ empruntées par un plus court chemin entre u et v (*i.e.* $0 \leqslant \varepsilon_{12} \leqslant 1$). De manière générale, nous définissons :

$$\begin{aligned}
&\varepsilon_{13} \text{ tel que } d(u, x) = d_1 + \varepsilon_{13} + d_3,\\
&\varepsilon_{14} \text{ tel que } d(u, y) = d_1 + \varepsilon_{14} + d_4,\\
&\varepsilon_{23} \text{ tel que } d(v, x) = d_2 + \varepsilon_{23} + d_3,\\
&\varepsilon_{24} \text{ tel que } d(v, y) = d_2 + \varepsilon_{24} + d_4,\\
&\varepsilon_{34} \text{ tel que } d(x, y) = d_3 + \varepsilon_{34} + d_4.
\end{aligned}$$

Puisque x n'est pas un sommet à distance maximum de u, nous avons :

$$d(u, v) > d(u, x) \qquad \Rightarrow \qquad d_1 + \varepsilon_{12} + d_2 > d_1 + \varepsilon_{13} + d_3$$

De même $exc(v)$ n'est pas maximum :

$$d(x, y) > d(v, y) \qquad \Rightarrow \qquad d_3 + \varepsilon_{34} + d_4 > d_2 + \varepsilon_{24} + d_4$$

Ces deux inégalités impliquent : $\quad d_3 + \varepsilon_{34} - \varepsilon_{24} > d_2 > d_3 + \varepsilon_{13} - \varepsilon_{12}$

d'où : $\quad \varepsilon_{34} - \varepsilon_{24} = 1$, $\varepsilon_{13} - \varepsilon_{12} = -1$ et $d_2 = d_3$

Autrement dit : $\quad \varepsilon_{34} = \varepsilon_{12} = 1$ et $\varepsilon_{24} = \varepsilon_{13} = 0$.

De même, en utilisant : $\quad d(u, v) > d(u, y)$ et $d(x, y) > d(v, x)$

Nous montrons que : $\quad d_4 + \varepsilon_{34} - \varepsilon_{23} > d_2 > d_4 + \varepsilon_{14} - \varepsilon_{12}$

D'où : $\quad 1 - \varepsilon_{23} = 1$, $\quad \varepsilon_{14} - 1 = -1 \quad$ et $\quad d_2 = d_4$

Nous obtenons donc : $\quad \varepsilon_{23} = \varepsilon_{14} = 0$.

Enfin nous savons que $d(u, v) < d(x, y)$, donc $d_1 + 1 + d_2 < d_3 + 1 + d_4$. Comme $d_2 = d_4$, nous avons finalement $d_1 < d_3$.

Pour résumer, nous avons prouvé les résultats suivants :

$$\varepsilon_{34} = \varepsilon_{12} = 1, \quad \varepsilon_{24} = \varepsilon_{13} = \varepsilon_{14} = \varepsilon_{23} = 0$$
$$d_1 < d_2 = d_3 = d_4$$

Fait 1 *Si $\varepsilon_{34} = 1$ et $d_1 < d_2 = d_3 = d_4$, alors il n'existe pas d'ordre Lex-BFS σ_1 terminant par le sommet u.*

Preuve : Puisque $\varepsilon_{34} = 1$, un plus court chemin P_{xy} entre x et y utilise une arête de la clique maximale $\mathcal{C}$. Soit (a_3, a_4) cette arête (*i.e.* $P_{xy} = [x, \ldots, a_3, a_4, \ldots, y]$. Notons que a_3 (*resp.* a_4) n'appartient à aucune clique maximale entre $\mathcal{C}$ et la clique contenant y (*resp.* x) sur la x, y-chaîne de cliques de l'arbre $\mathcal{T}$ (sinon $\varepsilon_{34} \neq 1$).

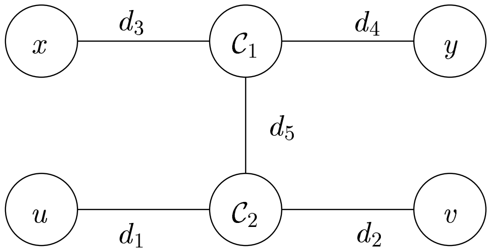
Nous notons a'_3 (*resp.* a'_4) le voisin de a_3 (*resp.* a_4) différent de a_4 (*resp.* a_3) sur P_{xy}. Supposons que a_3 soit numéroté avant a_4 dans l'ordre σ_1 (le cas inverse est équivalent). Le corollaire 3.8 montre que les sommets de toutes les cliques contenant l'arête (a_3, a_4) doivent être numérotés avant a'_4. Soit S l'ensemble des sommets numérotés lorsque toutes ces cliques sont visitées. La clique maximale $\mathcal{C}$ est donc visitée mais aucune clique maximale de la x, y-chaîne de cliques entre $\mathcal{C}$ et la clique contenant y n'est visitée. Nous avons donc :

$$d(S, u) \leqslant d(\mathcal{C}, u) = d_1 \qquad \text{et} \qquad d(S, y) = d(\mathcal{C}, y) = d_4$$

Or $d_1 < d_4$, le lemme des distances (lemme 2.8) montre que dans ces conditions σ_1 numérote u avant y : contradiction, nous avons montré que σ_1 ne peut pas finir sur u. □

L'hypothèse initiale était qu'il n'existe pas de sommet z d'excentricité maximum tel que $d(u, z) = excentricit(u)$. Le fait 1 montre que dans ce cas, il n'existe pas d'ordre Lex-BFS terminant sur u : contradiction, donc un tel sommet z existe.

II. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est vide :



Précisons nos notations : $d_1 = d(u, \mathcal{C}_2)$, $d_2 = d(v, \mathcal{C}_2)$, $d_3 = d(x, \mathcal{C}_1)$, $d_4 = d(y, \mathcal{C}_1)$ et $d_5 = d(\mathcal{C}_1, \mathcal{C}_2)$. Par exemple, $d(u, x) \geqslant d_1 + d_5 + d_3$. Il n'y a pas nécessairement égalité. Cela dépend du nombre d'arêtes qu'un plus court chemin entre u et x utilise dans $\mathcal{C}_1$ et $\mathcal{C}_2$. Nous noterons ce nombre ε_{13}. Il peut valoir 0, 1 ou 2. En revanche, $d(u, v) = d_1 + \varepsilon_{12} + d_2$ où ε_{12} est le nombre d'arêtes de $\mathcal{C}_2$ utilisées. Ici, ε_{12} peut valoir 0 ou 1.

Nous obtenons de manière générale :

$$\begin{array}{r}
\varepsilon_{13} \in [0,2] \text{ tel que } d(u,x) = d_1 + \varepsilon_{13} + d_5 + d_3 \\
\varepsilon_{14} \in [0,2] \text{ tel que } d(u,y) = d_1 + \varepsilon_{14} + d_5 + d_4 \\
\varepsilon_{23} \in [0,2] \text{ tel que } d(v,x) = d_2 + \varepsilon_{23} + d_5 + d_3 \\
\varepsilon_{24} \in [0,2] \text{ tel que } d(v,y) = d_2 + \varepsilon_{24} + d_5 + d_4 \\
\varepsilon_{12} \in [0,1] \text{ tel que } d(u,v) = d_1 + \varepsilon_{12} + d_2 \\
\varepsilon_{34} \in [0,1] \text{ tel que } d(x,y) = d_3 + \varepsilon_{34} + d_4
\end{array}$$

Puisque x n'est pas un sommet à distance maximum de u, nous avons :

$$d(u,v) > d(u,x) \quad \Rightarrow \quad d_1 + \varepsilon_{12} + d_2 > d_1 + \varepsilon_{13} + d_5 + d_3$$

Par contre $exc(v)$ n'est pas maximum :

$$d(x,y) > d(v,y) \quad \Rightarrow \quad d_3 + \varepsilon_{34} + d_4 > d_2 + \varepsilon_{24} + d_5 + d_4$$

Ces deux inégalités impliquent : $d_3 - d_5 + \varepsilon_{34} - \varepsilon_{24} > d_2 > d_3 + d_5 + \varepsilon_{13} - \varepsilon_{12}$

Or $\varepsilon_{34} - \varepsilon_{24}$ vaut au plus 1 et $\varepsilon_{13} - \varepsilon_{12}$ vaut au minimum -1 :

$$\Rightarrow \quad \varepsilon_{34} - \varepsilon_{24} = 1 \quad \text{et} \quad \varepsilon_{13} - \varepsilon_{12} = -1$$
$$\Rightarrow \quad d_5 = 0 \quad \text{et} \quad d_2 = d_3$$

Nous obtenons donc : $\varepsilon_{34} = \varepsilon_{12} = 1 \quad \text{et} \quad \varepsilon_{24} = \varepsilon_{13} = 0.$

De même, en utilisant : $d(u,v) > d(u,y) \quad \text{et} \quad d(x,y) > d(v,x)$

Nous pouvons montrer que : $d_2 + \varepsilon_{34} - \varepsilon_{14} > d_4 > d_2 + \varepsilon_{23} - \varepsilon_{12}$

D'où : $d_2 = d_4 \quad \text{et} \quad \varepsilon_{14} = \varepsilon_{23} = 0$

Enfin : $d(u,v) < d(x,y) \quad \Rightarrow \quad d_1 + 1 + d_2 < d_3 + 1 + d_4 \quad \Rightarrow \quad d_1 < d_3$

Pour résumer, nous avons prouvé les résultats suivants :

$$\varepsilon_{34} = \varepsilon_{12} = 1, \quad \varepsilon_{24} = \varepsilon_{13} = \varepsilon_{14} = \varepsilon_{23} = 0$$
$$d_5 = 0 \quad \text{et} \quad d_1 < d_2 = d_3 = d_4$$

Fait 2 *Si $\varepsilon_{34} = 1$ et $d_1 < d_2 = d_3 = d_4$, alors il n'existe pas d'ordre Lex-BFS σ_1 terminant par le sommet u.*

Preuve : Puisque $\varepsilon_{34} = 1$, un plus court chemin P_{xy} entre x et y utilise une arête de la clique maximale $\mathcal{C}_1$. Soit (a_3, a_4) cette arête (*i.e.* $P_{xy} =$

$[x, \ldots, a_3, a_4, \ldots, y]$. Notons que a_3 (*resp.* a_4) n'appartient à aucune clique maximale entre $\mathcal{C}_1$ et la clique contenant y (*resp.* x) sur la x,y-chaîne de cliques de l'arbre $\mathcal{T}$ (sinon $\varepsilon_{34} \neq 1$).

De plus, nous avons $\varepsilon_{13} = 0$ et $\varepsilon_{24} = 0$. Ces deux valeurs impliquent l'existence de deux sommets différents (car $\varepsilon_{34} = 1$), $a_{13} \in \mathcal{C}_1 \cap \mathcal{C}_2$ et $a_{24} \in \mathcal{C}_1 \cap \mathcal{C}_2$ tels que $d(u,x) = d(u,a_{13}) + d(a_{13},x) = d_1 + d_3$ et $d(v,y) = d(v,a_{24}) + d(a_{24},y) = d_2 + d_4$. Ainsi a_{13} peut jouer le rôle de a_3 et a_{24} celui de a_4.

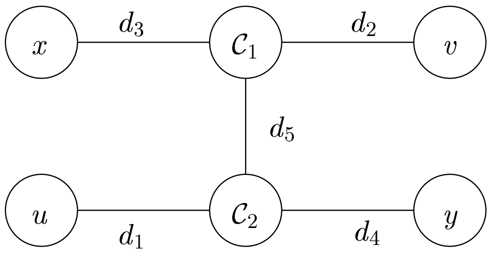
Nous notons a'_3 (*resp.* a'_4) le voisin de a_3 (*resp.* a_4) différent de a_4 (*resp.* a_3) sur P_{xy}. Supposons que a_3 soit numéroté avant a_4 dans l'ordre σ_1 (le cas inverse est équivalent). Le corollaire 3.8 montre que les sommets de toutes les cliques contenant l'arête (a_3, a_4) doivent être numérotés avant a'_4. Soit S l'ensemble des sommets numérotés lorsque toutes ces cliques sont visitées. Les cliques maximales $\mathcal{C}_1$ et $\mathcal{C}_2$ sont donc visitées mais aucune clique maximale (exceptée $\mathcal{C}_1$) de la x,y-chaîne de clique entre $\mathcal{C}_2$ et la clique contenant y n'est visitée. Nous avons donc :

$$d(S,u) \leqslant d(\mathcal{C}_2,u) = d_1 \qquad \text{et} \qquad d(S,y) = d(\mathcal{C}_1,y) = d_4$$

Or $d_1 < d_4$, le lemme des distances (lemme 2.8) montre que, dans ces conditions, σ_1 numérote u avant y : contradiction, nous avons montré que σ_1 ne peut pas finir sur u. □

Puisque qu'il n'existe pas d'ordre Lex-BFS terminant sur u, nous en déduisons qu'il doit exister parmi les sommets à distance maximum de u, un sommet z d'excentricité maximum.

III. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est une chaîne de cliques entre $\mathcal{C}_1$ et $\mathcal{C}_2$:



Adaptons nos notations à la nouvelle configuration :

$$\varepsilon_{12} \in [0,2] \text{ tel que } d(u,x) = d_1 + \varepsilon_{12} + d_5 + d_2$$
$$\varepsilon_{13} \in [0,2] \text{ tel que } d(u,v) = d_1 + \varepsilon_{13} + d_5 + d_3$$

$$\begin{aligned}
\varepsilon_{24} \in [0,2] &\text{ tel que } d(v,y) = d_2 + \varepsilon_{24} + d_5 + d_4\\
\varepsilon_{34} \in [0,2] &\text{ tel que } d(x,y) = d_3 + \varepsilon_{34} + d_5 + d_4\\
\varepsilon_{14} \in [0,1] &\text{ tel que } d(u,y) = d_1 + \varepsilon_{14} + d_2\\
\varepsilon_{23} \in [0,1] &\text{ tel que } d(x,v) = d_3 + \varepsilon_{23} + d_2
\end{aligned}$$

Nous avons à nouveau :

$$\begin{aligned}
d(u,v) > d(u,x) \quad &\Rightarrow \quad d_1 + \varepsilon_{12} + d_5 + d_2 > d_1 + \varepsilon_{13} + d_5 + d_3\\
d(x,y) > d(v,y) \quad &\Rightarrow \quad d_3 + \varepsilon_{34} + d_5 + d_4 > d_2 + \varepsilon_{24} + d_5 + d_4
\end{aligned}$$

Nous obtenons donc : $\quad d_3 + (\varepsilon_{34} - \varepsilon_{24}) > d_2 > d_3 - (\varepsilon_{12} - \varepsilon_{13}) \qquad$ **(1)**

Montrons que $\varepsilon_{34} - \varepsilon_{24} \leqslant 1$:

Supposons que $\varepsilon_{34} - \varepsilon_{24} = 0$ (*i.e.* $\varepsilon_{34} = 2$ et $\varepsilon_{24} = 0$). Alors $\varepsilon_{24} = 0$ implique l'existence de deux sommets $t \in \mathcal{C}_2$ et $t' \in \mathcal{C}_1$ tels que $d(t,t') = d_5$ et $d(t,v) = d_2$, $d(t',y) = d_4$. Dans ce cas $\varepsilon_{34} \leqslant 1$ (S'il n'existe pas $z \in C_1$ tel que $d(z,t) = d_5$ et $d(x,z) = d_3$, alors $\varepsilon_{34} = 1$. Si un tel sommet z existe, alors $\varepsilon_{34} = 0$). Donc $\varepsilon_{34} \neq 2$: contradiction.

La même démonstration peut-être faite pour ε_{12} et ε_{13}. Nous obtenons donc :

$$\varepsilon_{34} - \varepsilon_{24} \leqslant 1 \quad \text{et} \quad \varepsilon_{12} - \varepsilon_{13} \leqslant 1$$

L'inégalité (1) implique alors que :

$$\begin{aligned}
\varepsilon_{34} - \varepsilon_{24} = 1 \quad &\Rightarrow \quad \varepsilon_{34} = 1 \text{ et } \varepsilon_{24} = 0\\
\varepsilon_{12} - \varepsilon_{13} = 1 \quad &\Rightarrow \quad \varepsilon_{12} = 1 \text{ et } \varepsilon_{13} = 0\\
&\Rightarrow \quad d_2 = d_3
\end{aligned}$$

Montrons que $d_5 = 0$:

Supposons que $d_5 > 0$. Rappelons que $\varepsilon_{24} = 0$ implique l'existence de deux sommets $t \in \mathcal{C}_1$ et $t' \in \mathcal{C}_2$ tels que $d(t,t') = d_5$ et $d(t,v) = d_2$, $d(t',y) = d_4$. De même $\varepsilon_{13} = 0$ implique l'existence de deux sommets $z \in \mathcal{C}_1$ et $z' \in \mathcal{C}_2$ tels que $d(z,z') = d_5$ et $d(z,x) = d_3$, $d(z',u) = d_1$. Or $\varepsilon_{34} = 1$ implique que $d(z,t') > d_5$ donc $z' \neq t'$ et $\varepsilon_{12} = 1$ implique que $d(z',t) > d_5$ donc $z \neq t$. Soient $P_{tt'}$ et Pzz' deux plus courts chemins respectivement entre t, t' et z, z'. Considérons le cycle induit par les sommets de $P_{tt'}$ et Pzz'. Comme $d_5 > 0$, il existe au moins un voisin $a \neq t$ de t' sur $P_{tt'}$ et un voisin $b \neq z$ de z' sur $P_{zz'}$. Donc la longueur du cycle est supérieure à 4. Il doit posséder des cordes et en particulier il en existe une incidente soit à z' soit à t'. Sans perte de généralité, choisissons z'. Puisque $P_{tt'}$ et $P_{zz'}$ sont des chemins de distance minimum entre $\mathcal{C}_1$ et $\mathcal{C}_2$, le seul voisin de z' possible différent de t' et b est a. Alors $d(z',t) = d_5$ et $\varepsilon_{12} = 0$: contradiction, donc $d_5 = 0$.

Nous avons donc montré jusqu'ici que :

$$\varepsilon_{13} = \varepsilon_{24} = 0 \qquad \varepsilon_{12} = \varepsilon_{34} = 1 \qquad d_2 = d_3 \quad \text{et} \quad d_5 = 0$$

Intéressons nous aux valeurs de ε_{14} et ε_{23} :

$$d(u,v) > d(u,y) \quad \Rightarrow \quad d_1 + 1 + d_2 > d_1 + \varepsilon_{14} + d_4$$
$$d(x,y) > d(x,v) \quad \Rightarrow \quad d_3 + 1 + d_4 > d_3 + \varepsilon_{23} + d_2$$

Nous obtenons donc : $\quad d_4 + (1 - \varepsilon_{23}) > d_2 > d_4 - (1 - \varepsilon_{14})$

$$\Rightarrow \quad 1 - \varepsilon_{23} \geqslant 1 \qquad 1 - \varepsilon_{14} \geqslant 1 \quad \text{et} \quad d_2 = d_4$$
$$\Rightarrow \quad \varepsilon_{23} = \varepsilon_{14} = 0 \quad \text{et} \quad d_2 = d_4$$

Etudions maintenant les relations entre d_1 et d_2, d_3, d_4 :

$$d(u,v) < d(x,y) \quad \Rightarrow \quad d_1 + 1 + d_2 < d_3 + 1 + d_4 \quad \Rightarrow \quad d_1 < d_2$$

Pour résumer, nous avons montré les résultats suivant :

$$\varepsilon_{12} = \varepsilon_{34} = 1, \quad \varepsilon_{13} = \varepsilon_{24} = 0, \quad \varepsilon_{23} = \varepsilon_{14} = 0$$
$$d_5 = 0 \quad \text{et} \quad d_1 < d_2 = d_3 = d_4$$

Montrons qu'il existe un plus court chemin entre x et y utilisant une arête (a_3, a_4) de $\mathcal{C}_1 \cap \mathcal{C}_2$:

Nous avons $\varepsilon_{13} = 0$ et $\varepsilon_{24} = 0$. Ces deux valeurs impliquent l'existence de deux sommets $a_{13} \in \mathcal{C}_1 \cap \mathcal{C}_2$ et $a_{24} \in \mathcal{C}_1 \cap \mathcal{C}_2$ tels que $d(u,x) = d(u,a_{13}) + d(a_{13},x) = d_1 + d_3$ et $d(v,y) = d(v,a_{24}) + d(a_{24},y) = d_2 + d_4$. Notons que $\varepsilon_{34} = 1$ implique $a_{13} \neq a_{24}$.

Alors $a_3 = a_{13}$ et $a_4 = a_{24}$ et il existe un plus court chemin P_{xy} entre x et y qui utilise l'arête de G (a_3, a_4) appartenant à $\mathcal{C}_1 \cap \mathcal{C}_2$. Notons que a_3 (*resp.* a_4) n'appartient à aucune clique maximale entre $\mathcal{C}_2$ (*resp.* C_1) et la clique contenant y (*resp.* x) sur la x,y-chaîne de cliques de l'arbre $\mathcal{T}$ (sinon $\varepsilon_{34} \neq 1$).

Fait 3 *S'il existe un plus court chemin entre x et y utilisant une arête (a_3, a_4) de $\mathcal{C}_1 \cap \mathcal{C}_2$ et $d_1 < d_2 = d_3 = d_4$, alors il n'existe pas d'ordre Lex-BFS σ_1 terminant par le sommet u.*

Preuve : Nous notons a'_3 (*resp.* a'_4) le voisin de a_3 (*resp.* a_4) différent de a_4 (*resp.* a_3) sur P_{xy}. Supposons que a_3 soit numéroté avant a_4 dans l'ordre σ_1 (le cas inverse est équivalent). Le corollaire 3.8 montre que les sommets de toutes les cliques contenant l'arête (a_3, a_4) doivent être numérotés avant a'_4. Soit S l'ensemble des sommets numérotés lorsque toutes ces cliques sont visitées. Les cliques maximales $\mathcal{C}_1$ et $\mathcal{C}_2$ sont donc visitées mais aucune clique maximale de la x,y-chaîne de cliques entre $\mathcal{C}_2$ et la clique contenant y n'est visitée. Nous avons donc :

$$d(S,u) \leqslant d(\mathcal{C}_2,u) = d_1 \qquad \text{et} \qquad d(S,y) = d(\mathcal{C}_1,y) = d_4$$

Or $d_1 < d_4$, le lemme des distances (lemme 2.8) montre que dans ces conditions σ_1 numérote u avant y : contradiction, nous avons montré que σ_1 ne peut pas finir sur u. □

Puisque qu'il n'existe pas d'ordre Lex-BFS terminant sur u, on en déduit qu'il doit exister parmi les sommets à distance maximum de u, un sommet z d'excentricité maximum.

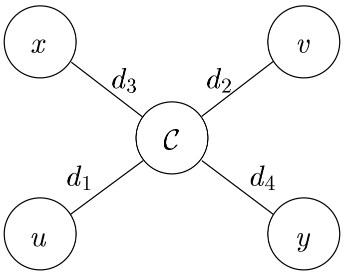
Finalement, nous avons montré que, dans toute les configurations possibles entre les chaînes de cliques $\mathcal{P}_{xy}$ et $\mathcal{P}_{uv}$, il est impossible pour σ_1 de finir sur u. Or par hypothèse, u est un sommet tel qu'il n'existe pas, dans l'ensemble des sommets à distance maximum de u, un sommet z d'excentricité maximum pour G. Donc si u est le dernier sommet numéroté par σ_1, un ordre Lex-BFS, il existe z un sommet de G tel $d(u,z) = exc(u)$ et $exc(z) = diam(G)$. □

Nous pouvons maintenant démontrer le résultat principal de cette section :

Théorème 3.11 *Soit G un graphe triangulé. Si l'heuristique 7 ne trouve pas un sommet v d'excentricité maximum, alors $exc(v) = diam(G) - 1$ et il existe un 3-soleil ou un 4-soleil.*

Preuve : Soit u le dernier sommet numéroté par un parcours Lex-BFS σ_1. Supposons qu'un second parcours Lex-BFS σ_2 commençant en u termine sur le sommet v qui n'est pas d'excentricité maximum. D'après le théorème 3.10, il existe un sommet x à distance maximum de u et d'excentricité maximum. Donc $d(u,v) = d(u,x)$. Soit $\mathcal{T}$ l'arbre de cliques associé à l'ordre σ_2. Le schéma de cette preuve sera exactement le même que celui de la preuve du théorème 3.10. Nous reprendrons donc les mêmes notations.

I. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est une clique $\mathcal{C}$:
Soit $\mathcal{T}_u$ (*resp.* $\mathcal{T}_v, \mathcal{T}_x$ et $\mathcal{T}_y$) le sous-arbre de $\mathcal{T}$ contenant u (*resp.* v, x et y) obtenu en supprimant la clique $\mathcal{C}$.



Fait 4 *Si v n'est pas un sommet d'excentricité maximum, alors $exc(v) = diam(G) - 1$.*

Preuve : Puisque les cliques des sous-arbres $\mathcal{T}_x, \mathcal{T}_y$ et $\mathcal{T}_v$ sont visitées après $\mathcal{C}$, pour que σ_2 finisse sur le sommet v, il faut que : $d_2 \geqslant d_3$ et $d_2 \geqslant d_4$

Or v n'est pas un sommet d'excentricité maximum :

$$d(x,y) > d(x,v) \quad \Rightarrow \quad d_3 + \varepsilon_{34} + d_4 > d_3 + \varepsilon_{23} + d_2$$

Alors $d_2 \geqslant d_4$ et $0 \leqslant \varepsilon_{34}, \varepsilon_{23} \leqslant 1 \quad \Rightarrow \quad \varepsilon_{34} = 1, \quad \varepsilon_{23} = 0$ et $d_2 = d_4$

De même $d(x,y) > d(y,v) \quad \Rightarrow \quad \varepsilon_{24} = 0$ et $d_2 = d_3$

Nous pouvons donc conclure dès maintenant que : $e(v) \geqslant d(v,y) = d_2 + d_4$

Or $d_2 + d_4 = d(x,y) - 1 = diam(G) - 1$ et par hypothèse, v n'est pas un sommet d'excentricité maximum, nous avons donc : $e(v) = d(v,y) = diam(G) - 1$

□

Il nous reste donc à montrer que le graphe G contient un 3-soleil ou un 4-soleil. Etudions plus en détails les relations entre les distances d_1, d_2, d_3, d_4 et les différents ε. Pour l'instant nous savons donc que :

$$\varepsilon_{23} = \varepsilon_{24} = 0, \quad \varepsilon_{34} = 1 \quad \text{et} \quad d_2 = d_3 = d_4$$

Les sommets u et v ne sont pas d'excentricité maximum :

$$d(u,v) < d(x,y) \quad \Rightarrow \quad d_1 + \varepsilon_{12} + d_3 < d_3 + 1 + d_4 \quad \Rightarrow \quad d_1 \leqslant d_2 = d_3 = d_4$$

Si $d_1 < d_2 = d_3 = d_4$, le fait 1 montre que σ_1 ne termine pas sur le sommet u. Nous devons donc avoir : $d_1 = d_2 = d_3 = d_4$ et $\varepsilon_{12} = 0$

Comme v et x sont des sommets à distance maximum de u :

$$d(u,v) = d(u,x) \quad \Rightarrow \quad d_1 + d_2 = d_1 + \varepsilon_{13} + d_3 \quad \Rightarrow \quad \varepsilon_{13} = 0$$

Enfin, σ_2 numérote y avant v :

$$d(u,v) \geqslant d(u,y) \quad \Rightarrow \quad d_1 + d_2 \geqslant d_1 + \varepsilon_{14} + d_3 \quad \Rightarrow \quad \varepsilon_{14} = 0$$

Finalement, nous obtenons : $\varepsilon_{23} = \varepsilon_{24} = \varepsilon_{13} = \varepsilon_{14} = \varepsilon_{12} = 0$

et : $\varepsilon_{34} = 1 \quad d_1 = d_2 = d_3 = d_4$

Fait 5 *Le graphe G contient un 3-soleil ou un 4-soleil.*

Preuve : Soient $N_c(x)$ les sommets z de $\mathcal{C}$ tels $d(x,z) = d(x,\mathcal{C})$. Nous définissons de la même manière $N_c(y), N_c(u)$ et $N_c(v)$. Notons que $N_c(x) \subseteq$

S_x où S_x est le séparateur minimal connectant $\mathcal{C}$ à $\mathcal{T}_x$, de même pour u, v et y. Nous avons :

$$\begin{array}{lcl}
\varepsilon_{34} = 1 & \Rightarrow & N_c(x) \cap N_c(y) = \emptyset \\
\varepsilon_{23} = 0 & \Rightarrow & \exists a_{23} \in (N_c(x) \cap N_c(v)) \setminus N_c(y) \\
\varepsilon_{24} = 0 & \Rightarrow & \exists a_{24} \in (N_c(y) \cap N_c(v)) \setminus N_c(x) \\
\varepsilon_{13} = 0 & \Rightarrow & \exists a_{13} \in (N_c(x) \cap N_c(u)) \setminus N_c(y) \\
\varepsilon_{14} = 0 & \Rightarrow & \exists a_{14} \in (N_c(y) \cap N_c(u)) \setminus N_c(x)
\end{array}$$

Montrons qu'il existe $a_{13} \notin N_c(v)$ et $a_{14} \notin N_c(v)$:

Remarquons d'abord que $a_{34} = 1$ implique que $a_{13} \neq a_{14}$. Soit S l'ensemble des sommets numérotés par σ_2 lorsque $N_c(u)$ est visité. A cette étape, nous avons $d(S, v) = d_2$, $d(S, x) = d_3$ et $d(S, y) = d_4$. Les sommets x, y et v sont donc équidistants de S. Nous ne devons donc pas avoir $N_s(x)$ et $N_s(y)$ inclus dans $N_s(v)$ (sinon, le lemme d'inclusion, lemme 3.9, montre que σ_2 ne termine pas par v). Puisque $a_{13} \neq a_{14}$, nous avons :

$$N_c(x) \cap N_c(u) \nsubseteq N_c(v) \quad \text{et} \quad N_c(y) \cap N_c(x) \nsubseteq N_c(v)$$

Nous pouvons donc choisir $a_{12} \notin N_c(v)$ et $a_{13} \notin N_c(v)$.

Soient $\mathcal{S}_{xy}$ les sommets appartenant à un plus court chemin entre x et y et s le premier sommet de $\mathcal{S}_{xy}$ numéroté par σ_1. Alors s doit appartenir à la clique $\mathcal{C}$:

Supposons le contraire, alors soit $d(x, s) > d_3 + 1$, soit $d(y, s) > d_4 + 1$. Sans perte de généralité, supposons que $d(x, s) > d_3 + 1$. Alors le corollaire 3.5 montre que les sommets de $\mathcal{S}_{xy}$ à distance $d_3 + 1$ de x seront numérotés avant ceux à distance d_3. Or les sommets de $\mathcal{S}_{xy}$ à distance $d_3 + 1$ de x sont à distance d_4 de y. Et parmi ces sommets, il en existe à distance d_1 de u (a_{14} par exemple). Il existe donc une étape de σ_1 où $d(S, u) < d(S, x)$. Le lemme des distances (lemme 2.8) montre que x est numéroté après u : contradiction, donc $s \in \mathcal{C}$.

Soit w le premier sommet numéroté par σ_1. Puisque σ_1 visite d'abord la clique $\mathcal{C}$ sur la x, y-chaîne de cliques, pour que le dernier sommet numéroté de σ_1 soit u, il faut numéroter des sommets de $N_c(x), N_c(y)$ et $N_c(v)$ avant de numéroter des sommets de $N_c(u)$. Sinon u serait plus proche que l'un des trois sommets x, y, v de l'ensemble des sommets numérotés. Il doit donc exister dans $N_c(w)$ un sommet $w_3 \in N_c(x) \setminus N_c(u)$, $w_4 \in N_c(y) \setminus N_c(u)$ et $w_2 \in N_c(v) \setminus N_c(u)$.

1. Si $w_3 \in N_c(v)$ et $w_4 \in N_c(v)$, puisque nous avons montré l'existence de $a_{13} \notin N_c(v)$ et $a_{14} \notin N_c(v)$, les sommets a_{13}, a_{14}, w_3, w_4 forment la

clique d'un 4-soleil. Les sommets du stable sont u', v', x', y' des sommets appartenant à la composante $cc(u)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_u]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant u (*resp.* v, x, y).

2. Si un des deux sommets w_3 et w_4 n'appartient pas à $N_c(v)$: par exemple $w_3 \in N_c(v)$ et $w_4 \notin N_c(v)$. Puisque $\varepsilon_{24} = 0$, il existe $a_{24} \in N_c(v) \cap N_c(y)$. Notons que si $a_{24} \in N_c(w)$, nous appliquons le cas précédent, donc $a_{24} \notin N_c(w)$. De plus $\varepsilon_{34} = 1$ implique que $a_{24} \neq w_3$. Alors w_3, w_4, a_{24} forment la clique d'un 3-soleil. Les sommets du stable sont w', v', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(y)$) de $G[V \setminus S_w]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_y]$) contenant w (*resp.* v, y).
3. Si $w_3 \notin N_c(v)$ et $w_4 \notin N_c(v)$. Puisque $\varepsilon_{23} = 0$ et $\varepsilon_{24} = 0$, il existe $a_{23} \in N_c(v) \cap N_c(x)$ et $a_{24} \in N_c(v) \cap N_c(y)$. Si l'un ou l'autre de ces sommets appartient à $N_c(w)$, nous retrouvons un des deux cas précédents. Supposons donc que $a_{23} \notin N_c(w)$ et $a_{24} \notin N_c(w)$. Alors les sommets a_{23}, a_{24}, w_3, w_4 forment la clique d'un 4-soleil. Les sommets du stable sont w', v', x', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_w]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant w (*resp.* v, x, y).

□

Nous avons donc montré dans tous les cas de figure l'existence soit d'un 3-soleil, soit d'un 4-soleil. La figure 3.1 montre un exemple de graphe triangulé pour lequel l'heuristique ne trouve pas le sommet d'excentricité maximum.

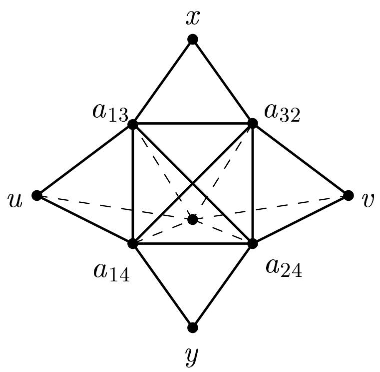
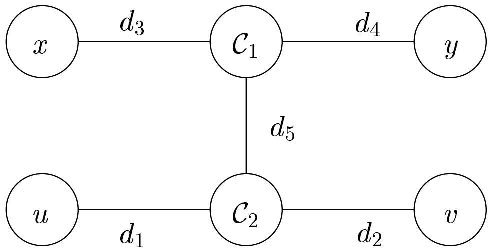


FIG. 3.1 – *Exemple de graphe triangulé où l'heuristique 7 peut échouer. Présence d'un 4-soleil et de 3-soleils*

II. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est vide :



Fait 6 *Si v n'est pas un sommet d'excentricité maximum alors $exc(v) = diam(G) - 1$.*

Preuve : Nous savons que σ_2 termine sur le sommet v, donc :
$d(u,v) \geqslant d(u,y) \quad \Rightarrow \quad d_1 + \varepsilon_{12} + d_2 \geqslant d_1 + d_5 + \varepsilon_{14} + d_4$

L'excentricité du sommet v n'est pas maximum, donc :
$d(x,y) > d(x,v) \quad \Rightarrow \quad d_3 + \varepsilon_{34} + d_4 > d_3 + d_5 + \varepsilon_{23} + d_2$

Ces deux inégalités impliquent : $d_4 + \varepsilon_{34} - (d_5 + \varepsilon_{23}) > d_2 \geqslant d_4 + d_5 + \varepsilon_{14} - \varepsilon_{12}$

Et donc : $\quad d_5 = 0 \quad$ et $\quad \varepsilon_{34} + \varepsilon_{12} > \varepsilon_{14} + \varepsilon_{23} \quad$ **(1)**

Puisque x et v sont des sommets à distance maximum de u, nous avons :
$d(u,v) = d(u,x) \quad \Rightarrow \quad d_1 + \varepsilon_{13} + d_3 = d_1 + \varepsilon_{12} + d_2$

Puisque v n'est pas d'excentricité maximum :

$$\begin{aligned} d(x,y) > d(y,v) \quad &\Rightarrow \quad d_3 + \varepsilon_{34} + d_4 > d_4 + \varepsilon_{24} + d_2 \\ &\Rightarrow \quad d_3 + \varepsilon_{34} + d_4 > d_4 + \varepsilon_{24} + (\varepsilon_{13} + d_3 - \varepsilon_{12}) \\ &\Rightarrow \quad \varepsilon_{34} + \varepsilon_{12} > \varepsilon_{24} + \varepsilon_{13} \quad \textbf{(2)} \end{aligned}$$

Pour satisfaire les inégalités (1) et (2), il est nécessaire que que $\varepsilon_{12} = 1$ ou/et $\varepsilon_{34} = 1$. Envisageons les différents cas :

1. **Supposons que** $\varepsilon_{12} = 1$ **et** $\varepsilon_{34} = 0$ **:** Cela implique que $\varepsilon_{13} = \varepsilon_{14} = \varepsilon_{23} = \varepsilon_{24} = 0$. Comme x et v sont des sommets à distance maximum de u, nous obtenons :
   $d(u,v) = d(u,x) \;\Rightarrow\; d_1 + 1 + d_2 = d_1 + d_3 \;\Rightarrow\; d_2 < d_3$
   Or avant de numéroter x et v, σ_2 visite les sommets de la clique $\mathcal{C}_2$. A ce stade, nous avons donc $d(v,S) < d(x,S)$, le lemme des distances (lemme 2.8) implique que v est numéroté avant x : contradiction.
2. **Supposons que** $\varepsilon_{12} = 1$ **et** $\varepsilon_{34} = 1$ **:** Donc d'après (1), nous devons avoir $\varepsilon_{13} = 0$ et/ou $\varepsilon_{24} = 0$. Les sommets x et v sont équidistants de u :
   $d(u,v) = d(u,x) \quad \Rightarrow \quad d_1 + 1 + d_2 = d_1 + \varepsilon_{13} + d_3 \quad$ **(3)**
   Si $\varepsilon_{13} = 0$, alors l'égalité (3) implique que : $\quad 1 + d_2 = d_3 \Rightarrow d_2 < d_3$

Soit S l'ensemble des sommets numérotés par σ_2 lorsque tous les sommets de $\mathcal{C}_2$ ont été numérotés. A cette étape, nous avons $d(v, S) = d_2 < d(x, S) = d_3$. Le lemme des distances (lemme 2.8) implique que v est numéroté avant x : contradiction. Nous devons donc avoir : $\varepsilon_{13} = 1$ et (à cause de (2)) $\varepsilon_{24} = 0$.

Dans ce cas, l'égalité (3) montre que : $\quad d_2 = d_3$

Puisque v n'est pas un sommet d'excentricité maximum :

$d(v, x) < d(x, y) \quad \Rightarrow \quad d_2 + \varepsilon_{23} + d_3 < d_3 + \varepsilon_{34} + d_4$

Or $\varepsilon_{34} = 1$, donc : $\quad d_2 + \varepsilon_{23} \leqslant d_4$ **(4)**

Supposons que $\varepsilon_{23} = 1$, alors : $\quad d_2 < d_4$

Soit S l'ensemble des sommets numérotés quand ceux de la clique $\mathcal{C}_2$ ont été numérotés par σ_2. Le lemme des distances (lemme 2.8) implique que $d(y, S) = d_4 \leqslant d_2 = d(v, S)$: impossible si $\varepsilon_{23} = 1$. Donc $\varepsilon_{23} = 0$ et alors (4) implique $d_2 = d_4$.

Alors, puisque u et v ne sont pas d'excentricité maximum :

$d(u, v) < d(x, y) \quad \Rightarrow \quad d_1 + 1 + d_2 < d_3 + 1 + d_4 \quad \Rightarrow \quad d_1 < d_3$

Jusqu'ici nous avons :

$$d_5 = 0 \text{ et } d_1 < d_2 = d_3 = d_4$$
$$\varepsilon_{34} = \varepsilon_{12} = \varepsilon_{13} = 1 \text{ et } \varepsilon_{23} = \varepsilon_{24} = 0$$

Alors l'inégalité (1) implique que : $\quad 0 \leqslant \varepsilon_{14} \leqslant 1$.

Montrons que l'ordre Lex-BFS σ_1 ne peut pas terminer sur v :

Soit S_{xy} l'ensemble des sommets appartenant à un plus court chemin entre x et y (S_{xy} est contenu dans la x, y-chaîne de cliques). Soit s le premier sommet de S_{xy} numéroté par σ_1. Montrons que $s \in \mathcal{C}_1$. Si ce n'est pas le cas, ou bien $d(s, x) > d_3 + 1$, ou $d(s, y) > d_4 + 1$. Nous notons $N_{C_1}(x)$ (*resp.* $N_{C_1}(y)$) l'ensemble de sommets de $\mathcal{C}_1$ à distance $_3$ de x (*resp.* y).

Supposons que $d(s, x) > d_3 + 1$. Soit S l'ensemble des sommets numérotés quand $N_{C_1}(y)$ est numéroté. Nous avons alors $d(u, S) \leqslant d_1 + \varepsilon_{14}$ et $d(x, S) = d_3 + 1$. Puisque $d_1 < d_3$, le lemme des distances (lemme 2.8) montre que c'est impossible pour σ_1 de terminer sur u.

Supposons que $d(s, y) > d_4 + 1$. Soit S l'ensemble des sommets numérotés quand $N_{C_1}(x)$ est numéroté. Nous avons alors $d(u, S) \leqslant d_1 + \varepsilon_{13} = d_1 + 1$ et $d(y, S) = d_4 + 1$. Puisque $d_1 < d_4$, le lemme des distances (lemme 2.8) montre que c'est impossible pour σ_1 de terminer sur u.

Donc les premiers sommets de S_{xy} numérotés sont ceux de $\mathcal{C}_1$. Soit S l'ensemble des sommets numérotés quand ceux de $\mathcal{C}_1$ le sont. Nous avons alors $d(u, S) \leqslant d_1 + eps_{14}$ et $d(x, S) = d_3$. Le lemme des distances implique alors que :

$$d_3 = d(x, S) \leqslant d(u, S) \leqslant d_1 + \varepsilon_{14}$$

Or $d_1 < d_3$, nous devons donc avoir $\varepsilon_{14} = 1$ et $d_1 + 1 = d_3 = d_2 = d_4$.

Notons que tous les sommets de $N_{C_1}(v)$ (*i.e.* ensemble des sommets de $\mathcal{C}_1$ à distance minimum de v, c'est-à-dire d_1 car $\varepsilon_{23} = \varepsilon_{24} = 0$) sont à distance $d_1 + 1$ de u. Donc $N_{C_1}(v) \subseteq N_{C_1}(u)$.

De plus avant de numéroter les sommets de $\mathcal{C}_1$, σ_1 n'a pas pu numéroter un sommet à distance inférieure à d_2 de v. Sinon des sommets à distance d_1 de u seraient numérotés avant des sommets à distance $d_3 - 1$ de x. Le lemme des distances (lemme 2.8) montre que ce n'est pas possible. Donc pour que σ_1 termine sur u, l'ordre lexicographique impose que $N_{C_1}(v) = N_{C_1}(u)$.

Considérons alors Σ_2. Rappelons que $N_{C_1}(v) \subset \mathcal{C}_1 \cap \mathcal{C}_2$. Soient S les sommets numérotés lorsque les sommets de $\mathcal{C}_2$ le sont. Puisque $\varepsilon_{34} = 1$, $N_S(x) \subsetneq N_S(v)$, alors v est numéroté avant x : contradiction.

3. $\varepsilon_{12} = 0$ et $\varepsilon_{34} = 1$: Comme dans le cas précédent, nous avons $\varepsilon_{13} = \varepsilon_{14} = \varepsilon_{23} = \varepsilon_{24} = 0$. Montrons que $exc(v) = diam(G) - 1$. Pour cela, étudions les relations entre les distances d_1, d_2, d_3, d_4. Les sommets x et v sont équidistants de u :

   $d(u, v) = d(u, x) \quad \Rightarrow \quad d_1 + d_2 = d_1 + d_3 \quad \Rightarrow \quad d_2 = d_3$

   L'excentricité du sommet v n'est pas maximum :

   $d(x, y) > d(x, v) \quad \Rightarrow \quad d_3 + 1 + d_4 > d_3 + d_2 \quad \Rightarrow \quad d_4 \geqslant d_2$

   Dans σ_2, le sommet y est numéroté avant v :

   $d(u, v) \geqslant d(u, y) \quad \Rightarrow \quad d_1 + d_2 \geqslant d_1 + d_4 \quad \Rightarrow \quad d_2 \geqslant d_4$

   Nous obtenons donc : $d_2 = d_3 = d_4$

   Nous pouvons donc conclure que : $e(v) \geqslant d(v, y) = d_2 + d_4$

   Or $d_2 + d_4 = d(x, y) - 1 = diam(G) - 1$, et par hypothèse, v n'est pas un sommet d'excentricité maximum. Donc : $e(v) = d(v, y) = diam(G) - 1$

□

Il nous reste donc à montrer que dans ce cas le graphe G contient un 3-soleil ou un 4-soleil. Pour cela nous devons étudier un peu plus la relation entre d_1 et les distances d_2, d_3, d_4. Puisque u et v ne sont pas des sommets d'excentricité maximum :

$d(x, y) > d(u, v) \quad \Rightarrow \quad d_3 + 1 + d_4 > d_1 + d_2 \quad \Rightarrow \quad d_1 \leqslant d_3 = d_2 = d_4$

Or $\varepsilon_{34} = 1$, si $d_1 < d_2 = d_3 = d_4$, le fait 1 peut s'appliquer (car $d(u, \mathcal{C}_1) = d_1$). Il montre que σ_1 ne peut pas terminer sur le sommet u. Nous devons donc avoir : $d_1 = d_2 = d_3 = d_4$

Rappelons que nous avions aussi :

$$\varepsilon_{12} = \varepsilon_{13} = \varepsilon_{14} = \varepsilon_{23} = \varepsilon_{24} = 0 \qquad \varepsilon_{34} = 1 \qquad \text{et} \qquad d_5 = 0$$

Fait 7 *Le graphe G contient un 3-soleil ou un 4-soleil.*

Preuve : Soient $N_{c_1}(x)$ les sommets z de $\mathcal{C}_1$ tels $d(x, z) = d(x, \mathcal{C})$. Nous définissons de la même manière $N_{c_1}(y), N_{c_1}(u)$ et $N_{c_1}(v)$. Notons que $N_{c_1}(x) \subseteq S_x$, de même pour u, v et y. Nous avons :

$$\begin{array}{lcl} \varepsilon_{13} = 0 & \Rightarrow & \exists a_{13} \in N_{c_1}(u) \cap N_{c_1}(x) \\ \varepsilon_{14} = 0 & \Rightarrow & \exists a_{14} \in N_{c_1}(u) \cap N_{c_1}(y) \\ \varepsilon_{23} = 0 & \Rightarrow & \exists a_{23} \in N_{c_1}(v) \cap N_{c_1}(x) \\ \varepsilon_{24} = 0 & \Rightarrow & \exists a_{24} \in N_{c_1}(v) \cap N_{c_1}(y) \end{array}$$

Montrons qu'il existe $a_{13} \notin N_c(v)$ et $a_{14} \notin N_c(v)$:

Soient $\mathcal{T}_x$ et $\mathcal{T}_y$ les sous-arbres de $\mathcal{T}$ obtenus en supprimant $\mathcal{C}_1$ contenant respectivement la clique de x et de y. En commençant par u, l'ordre σ_2 doit numéroter les sommets de $\mathcal{C}_1 \cap \mathcal{C}_2$ avant de visiter les sous-arbres $\mathcal{T}_x$ et $\mathcal{T}_y$. Soit S l'ensemble des sommets numérotés à cette étape.

Si nous ne pouvons pas choisir $a_{13} \notin N_{c_1}(v)$, alors $N_s(x) \subsetneq N_s(v)$. Donc σ_2 ne finit pas sur v à cause du lemme d'inclusion (lemme 3.9). De même, si nous ne pouvons pas trouver $a_{14} \notin N_{c_1}(v)$, alors $N_s(y) \subsetneq N_s(v)$. Donc σ_2 ne finit pas sur v à cause du lemme d'inclusion (lemme 3.9). Nous pouvons donc choisir $a_{13} \notin N_{c_1}(v)$ et $a_{14} \notin N_{c_1}(v)$.

Soient $\mathcal{S}_{xy}$ les sommets appartenant à un plus court chemin entre x et y. Et soit s le premier sommet de $\mathcal{S}_{xy}$ numéroté par σ_1. Alors s doit appartenir à la clique $\mathcal{C}_1$:

Si ce n'est pas le cas, alors nous avons $d(x, s) > d_3 + 1$ ou $d(y, s) > d_4 + 1$. Supposons que $d(x, s) > d_3 + 1$. Alors le corollaire 3.5 montre que les sommets de $\mathcal{S}_{xy}$ à distance $d_3 + 1$ de x seront numérotés avant ceux à distance d_3. Or les sommets de $\mathcal{S}_{xy}$ à distance $d_3 + 1$ de x sont à distance d_4 de y. Et parmi ces sommets il en existe à distance d_1 de u (car $\varepsilon_{14} = 0$). Il existe donc une étape de σ_1 où $d(S, u) < d(S, x)$. Le lemme des distances (lemme 2.8) montre que x est numéroté après u : contradiction, donc $s \in \mathcal{C}_1$.

Soit w le premier sommet numéroté par σ_1. Puisque la première clique de la x, y-chaîne de cliques visitée par σ_1 est $\mathcal{C}_1$, pour que le dernier sommet numéroté par σ_1 soit u, il faut numéroter des sommets de $N_{c_1}(x)$, $N_{c_1}(y)$ et $N_{c_1}(v)$ avant de numéroter des sommets de $N_{c_1}(u)$. Sinon u serait plus proche que l'un des trois sommets x, y, v de l'ensemble des sommets numérotés. Il doit donc exister dans $N_{c_1}(w)$ un sommet $w_3 \in N_{c_1}(x) \setminus N_{c_1}(u)$, $w_4 \in N_{c_1}(y) \setminus N_{c_1}(u)$ et $w_2 \in N_{c_1}(v) \setminus N_{c_1}(u)$.

1. Si $w_3 \in N_{c_1}(v)$ et $w_4 \in N_{c_1}(v)$, puisque nous avons montré l'existence de $a_{13} \notin N_{c_1}(v)$ et $a_{14} \notin N_{c_1}(v)$, les sommets a_{13}, a_{14}, w_3, w_4 forment la clique d'un 4-soleil. Les sommets du stable sont u', v', x', y' des sommets appartenant à la composante $cc(u)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_u]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant u (*resp.* v, x, y).

2. Si un des deux sommets w_3 et w_4 n'appartient pas à $N_{c_1}(v)$: par exemple, $w_3 \in N_{c_1}(v)$ et $w_4 \notin N_{c_1}(v)$. Puisque $\varepsilon_{24} = 0$, il existe $a_{24} \in N_{c_1}(v) \cap N_{c_1}(y)$. Notons que si $a_{24} \in N_{c_1}(w)$, nous appliquons le cas précédent, donc $a_{24} \notin N_{c_1}(w)$. Alors w_3, w_4, a_{24} forment la clique d'un 3-soleil. Les sommets du stable sont w', v', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(y)$) de $G[V \setminus S_w]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_y]$) contenant w (*resp.* v, y).

3. Si $w_3 \notin N_{c_1}(v)$ et $w_4 \notin N_{c_1}(v)$. Puisque $\varepsilon_{23} = 0$ et $\varepsilon_{24} = 0$, il existe $a_{23} \in N_{c_1}(v) \cap N_{c_1}(x)$ et $a_{24} \in N_{c_1}(v) \cap N_{c_1}(y)$. Si l'un ou l'autre de ces sommets appartient à $N_{c_1}(w)$, nous retrouvons un des deux cas précédents. Supposons donc que $a_{23} \notin N_{c_1}(w)$ et $a_{24} \notin N_{c_1}(w)$. Alors les sommets a_{23}, a_{24}, w_3, w_4 forment la clique d'un 4-soleil. Les sommets du stable sont w', v', x', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_w]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant w (*resp.* v, x, y).

$\square$

Nous avons donc montré dans tous les cas de figure, l'existence, soit d'un 3-soleil, soit d'un 4-soleil. La figure 3.1 montre un exemple de graphe triangulé pour lequel l'heuristique ne trouve pas le sommet d'excentricité maximum.

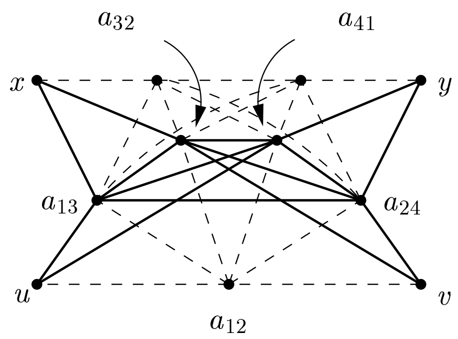
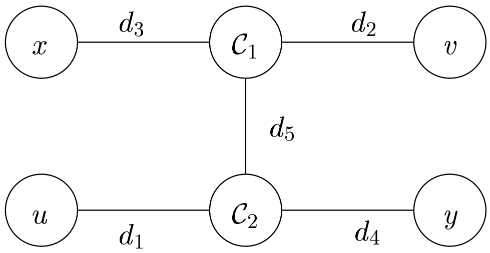


FIG. 3.2 – *Exemple de graphe triangulé où l'heuristique 7 peut échouer. Présence d'un 4-soleil.*

III. L'intersection entre $\mathcal{P}_{uv}$ et $\mathcal{P}_{xy}$ dans $\mathcal{T}$ est une chaîne de cliques entre $\mathcal{C}_1$ et $\mathcal{C}_2$:



Fait 8 *Si v n'est pas un sommet d'excentricité maximum alors $exc(v) = diam(G) - 1$.*

Preuve : L'excentricité du sommet v n'est pas maximum :

$$d(x.y) > d(v,y) \quad \Rightarrow \quad d_3 + d_5 + \varepsilon_{34} + d_4 > d_2 + d_5 + \varepsilon_{24} + d_4$$
$$\Rightarrow \quad d_3 + \varepsilon_{34} > d_2 + \varepsilon_{24}$$

De plus, il est impossible d'avoir $\varepsilon_{34} - \varepsilon_{24} = 2$. Il faudrait que $\varepsilon_{24} = 0$. Cela impliquerait l'existence de $z \in \mathcal{C}_2$ tel que $d(z,y) = d_4$ et $d(z,\mathcal{C}_1) = d_5$. Dans ce cas $\varepsilon_{34} \leqslant 1$ et donc la différence ne peut pas être 2. Nous obtenons donc :

$$\varepsilon_{34} - \varepsilon_{24} \geqslant 1 \quad \Rightarrow \quad d_3 + 1 > d_2 \quad \Rightarrow \quad d_3 \geqslant d_2$$

Pour que l'ordre σ_2 finisse par le sommet v, lorsque les sommets de $\mathcal{C}_1$ sont numérotés, on doit avoir (lemme 2.8) : $\quad d_3 \leqslant d_2$

Nous obtenons donc : $\quad d_3 = d_2 \quad$ *et* $\quad \varepsilon_{34} - \varepsilon_{24} = 1$

Supposons que $\varepsilon_{24} = 1$. Cela implique $\varepsilon_{34} = 2$ et donc :

$$\nexists z \in \mathcal{C}_1 \text{ tel que } d(z, \mathcal{C}_2) = d_5 \text{ et } d(z, x) = d_3$$
$$\nexists z' \in \mathcal{C}_2 \text{ tel que } d(z', \mathcal{C}_1) = d_5 \text{ et } d(z', y) = d_4$$

Pour que $\varepsilon_{24} = 1$, il doit alors exister un sommet $t \in \mathcal{C}_1$ tel que $d(t, \mathcal{C}_2) = d_5$ et $d(t, v) = d_2$. Dans l'ordre σ_2, les premiers sommets numérotés de $\mathcal{C}_1$ sont ceux dont la distance à $\mathcal{C}_2$ est égale à d_5. Soit S l'ensemble des sommets numérotés à cette étape. Nous avons alors $d(x, S) = d_3 + 1 > d_2 = d(v, S)$. C'est impossible (lemme 2.8).

Donc nous avons : $\varepsilon_{24} = 0$ *et* $\varepsilon_{34} = 1$

Nous allons montrer maintenant que $d_5 = 0$ *:*

Puisque $\varepsilon_{24} = 0$ implique : $\exists z \in \mathcal{C}_1$ tel que $d(z, \mathcal{C}_2) = d_5$ et $d(z, v) = d_2$
et $\exists z' \in \mathcal{C}_2$ tel que $d(z', \mathcal{C}_1) = d_5$ et $d(z', y) = d_4$

D'autre part, $\varepsilon_{34} = 1$ implique : $\nexists t \in N_{c_1}(x)$ tel que $d(t, z') = d_5$

Supposons que $\nexists t \in N_{c_1}(x)$ tel que $d(t, \mathcal{C}_2) = d_5$. Soit S l'ensemble des sommets quand σ_2 a fini de visiter $\mathcal{C}_2$. A cette étape, nous avons $d(S, x) = d_5 + 1 + d_3$ et $d(S, v) = d_5 + d_2$. Or nous avons montré que $d_2 = d_3$. Cela signifie que $d(S, x) > d(S, v)$. Alors d'après le lemme des distances (lemme 2.8), v est numéroté avant x : contradiction, donc : $\exists t \in N_{c_1}(x)$ tel que $d(t, \mathcal{C}_2) = d_5$

Soit donc $t' \in \mathcal{C}_2$ tel que $d(t, t') = d_5$. Notons que $\varepsilon_{34} = 1$ implique que $t' \notin N_{c_2}(y)$. Si $d_5 > 0$, le cycle constitué par deux plus courts chemins entre z, z' et t, t' doit être triangulé. La seule manière de respecter $d(z', t) > d_5$ est d'avoir $d(z, t') = d_5$.

Soit S est l'ensemble des sommets quand σ_2 a finit de visiter $\mathcal{C}_2$. Alors $d(z, t') = d_5$ implique que $N_s(x) \subsetneq N_s(v)$. Or d'après le lemme d'inclusion (lemme 3.9), c'est donc impossible de finir en v : contradiction.

Nous avons effectivement : $d_5 = 0$.

Rappelons que nous venons de montrer en même temps que :
$$\exists t \in N_{c_1}(x) \text{ tel que } d(t, \mathcal{C}_2) = d_5$$

Autrement dit, puisque $d_5 = 0$ implique $\mathcal{C}_1 \cap \mathcal{C}_2 \neq \emptyset$, il existe $t \in N_{c_1}(x) \cap \mathcal{C}_2$. Comme $\varepsilon_{34} = 1$, ce sommet n'appartient pas à $N_{c_2}(y)$. Nous nous servirons de cette propriété plus tard.

Etudions maintenant les valeurs de ε_{23}*,* d_2 *et* d_4 *:*

Considérons à nouveau l'ensemble S des sommets numérotés lorsque σ_2 a fini de visiter $\mathcal{C}_2$. A cette étape, nous avons $d(S, y) = d_4$ et $d(S, v) = d_2$. Afin de numéroter le sommet v en dernier, le lemme des distance (lemme 2.8) implique donc que : $d(S, v) \geqslant d(S, y) \quad \Rightarrow \quad d_2 \geqslant d_4$

Or le sommet v n'est pas un sommet d'excentricité maximum, donc :

$$d(x,y) > d(x,v) \quad \Rightarrow \quad d_3 + 1 + d_4 > d_3 + \varepsilon_{23} + d_2$$
$$\Rightarrow \quad 1 + d_4 - \varepsilon_{23} > d_2 \geqslant d_4$$

Nous devons donc avoir : $\varepsilon_{23} = 0$ *et* $d_2 = d_3 = d_4$

Rappelons que nous avions déjà : $d_5 = 0, \quad \varepsilon_{34} = 1$ *et* $\varepsilon_{24} = 0$

Nous pouvons donc conclure que : $e(v) \geqslant d(v,y) = d_2 + d_4$

Or $d_2 + d_4 = d(x,y) - 1 = diam(G) - 1$ et par hypothèse, v n'est pas un sommet d'excentricité maximum. Donc : $e(v) = d(v,y) = diam(G) - 1$ □

Il nous reste donc à montrer que dans ce cas le graphe G contient un 3-soleil ou un 4-soleil. Pour cela nous devons étudier un peu plus les relations entre les distances d_1, d_2, d_3, d_4 et les différents ε.

Puisque u et v ne sont pas des sommets d'excentricité maximum et $d_5 = 0$:
$d(x,y) > d(u,v) \quad \Rightarrow \quad d_3 + 1 + d_4 > d_1 + \varepsilon_{12} + d_3 \quad \Rightarrow \quad d_1 \leqslant d_2 = d_3 = d_4$

Pour l'instant, nous savons donc que :
$\varepsilon_{23} = \varepsilon_{24} = 0, \quad \varepsilon_{34} = 1, \quad d_5 = 0$ et $d_1 \leqslant d_2 = d_3 = d_4$

Pour appliquer le fait 3, il faut montrer qu'il existe un plus court chemin entre x et y utilisant une arête de $\mathcal{C}_1 \cap \mathcal{C}_2$:

- $\varepsilon_{24} = 0$ implique l'existence d'un sommet $a_{24} \in \mathcal{C}_1 \cap \mathcal{C}_2$ tel que $d(y,v) = d(y,a_{24}) + d(a_{24},v) = d_4 + d_2$. Et puisque $\varepsilon_{34} = 1$, $d(a_{24},x) = d_3 + 1$.
- Soit S l'ensemble des sommets numérotés lorsque σ_2 a fini de visiter la clique $\mathcal{C}_2$. Nous avons alors $d(S,v) = d(a_{24},v) = d_2$. Pour finir sur le sommet u, le lemme des distances impliques que $d(S,x) \leqslant d(S,v) = d_2 = d_3$. Or $d_3 = d(\mathcal{C}_1,x) \geqslant d(\mathcal{C}_2,x)$, donc il existe un sommet $a_3 \in \mathcal{C}_1 \cap \mathcal{C}_2$ tel que $d(a_3,x) = d_3$. Puisque $\varepsilon_{34} = 1$, les sommets a_3 et a_{24} sont différents.

L'arête (a_3, a_{24}) est donc une arête de $\mathcal{C}_1 \cap \mathcal{C}_2$ appartenant à un plus court chemin entre x et y. Le fait 3 montre alors que si $d_1 < d_2 = d_3 = d_4$, alors il n'existe pas d'ordre Lex-BFS terminant sur u : contradiction. Nous avons donc : $d_1 = d_2 = d_3 = d_4$

Dans ce cas, nous pouvons connaître les valeurs des ε restant. Puisque u et v ne sont pas des sommets d'excentricité maximum et $d_5 = 0$:

$$d(u,v) < d(x,y) \quad \Rightarrow \quad d_1 + \varepsilon_{12} + d_2 < d_3 + 1 + d_4 \quad \Rightarrow \quad \varepsilon_{12} = 0$$

Comme v et x sont des sommets à distance maximum de u :

$$d(u,v) = d(u,x) \quad \Rightarrow \quad d_1 + d_2 = d_1 + \varepsilon_{13} + d_3 \quad \Rightarrow \quad \varepsilon_{13} = 0$$

Enfin, σ_2 numérote y avant v:

$$d(u,v) \geqslant d(u,y) \quad \Rightarrow \quad d_1 + d_2 \geqslant d_1 + \varepsilon_{14} + d_3 \quad \Rightarrow \quad \varepsilon_{14} = 0$$

Et nous obtenons donc finalement: $d_5 = 0$ et $d_1 = d_2 = d_3 = d_4$

$$\varepsilon_{23} = \varepsilon_{24} = \varepsilon_{13} = \varepsilon_{14} = \varepsilon_{12} = 0 \quad \text{et} \quad \varepsilon_{34} = 1$$

Fait 9 *Il existe dans G un 3-soleil.*

Preuve : Nous notons $N_{c_2}(v)$ les sommets z de $\mathcal{C}_2$ tels $d(v,z) = d(v,\mathcal{C}_2)$ ($N_{c_2}(y)$ $N_{c_2}(x)$ et $N_{c_2}(u)$ sont définis de la même manière). Alors:

$\varepsilon_{13} = 0 \quad \Rightarrow \quad \exists a_{13} \in \mathcal{C}_1 \cap \mathcal{C}_2$ tel que $d(x, a_{13}) = d_3$ et $d(u, a_{13}) = d_1$

i.e. $a_{13} \in N_{c_2}(x) \cap N_{c_2}(u)$

$\varepsilon_{14} = 0 \quad \Rightarrow \quad \exists a_{14} \in \mathcal{C}_2$ tel que $d(y, a_{14}) = d_4$ et $d(u, a_{14}) = d_1$

i.e. $a_{14} \in N_{c_2}(y) \cap N_{c_2}(u)$

Remarquons que a_{14} n'appartient pas nécessairement à $\mathcal{C}_1 \cap \mathcal{C}_2$.

Soient $\mathcal{T}_y, \mathcal{T}_u$ et $\mathcal{T}_1$ les sous-arbres de $\mathcal{T}$ obtenu en supprimant $\mathcal{C}_2$ contenant respectivement la clique de y, de u et la clique $\mathcal{C}_1$. En commençant par u, l'ordre σ_2 doit numéroter les sommets de $\mathcal{C}_2$ avant de visiter les sous-arbres $\mathcal{T}_y$ et $\mathcal{T}_1$. Soit S l'ensemble des sommets numérotés à cette étape.

Si on ne peut pas choisir $a_{13} \notin N_{c_2}(v)$, alors $N_s(x) \subsetneq N_s(v)$. Donc σ_2 ne finit pas sur v à cause du lemme d'inclusion (lemme 3.9). De même, si on ne peut pas trouver $a_{14} \notin N_{c_2}(v)$, alors $N_s(y) \subsetneq N_s(v)$. Donc σ_2 ne finit pas sur v à cause du lemme d'inclusion (lemme 3.9).

Nous pouvons donc choisir $a_{13} \notin N_{c_2}(v)$ ($a_{13} \in \mathcal{C}_1 \cap \mathcal{C}_2$) et $a_{14} \notin N_{c_2}(v)$. Puisque $\varepsilon_{34} = 1$ ces deux sommets sont différents.

Soient $\mathcal{S}_{xy}$ les sommets appartenant à un plus court chemin entre x et y. Et soit s le premier sommet de $\mathcal{S}_{xy}$ numéroté par σ_1. Alors s doit appartenir à une clique entre $\mathcal{C}_1$ entre $\mathcal{C}_2$ dans $\mathcal{T}$:

Si ce n'est pas le cas, alors nous avons $d(x,v) > d_3 + 1$ ou $d(y,s) > d_4 + 1$. Supposons que $d(x,s) > d_3+1$. Alors le corollaire 3.5 montre que les sommets de $\mathcal{S}_{xy}$ à distance d_3+1 de x seront numérotés avant ceux à d_3. Or les sommets de $\mathcal{S}_{xy}$ à distance d_3+1 de x sont à distance d_4 de y. Et parmi ces sommets, il en existe à distance d_1 de u (car $\varepsilon_{14} = 0$). Il existe donc une étape de σ_1 où $d(S,u) < d(S,x)$. Le lemme des distances (lemme 2.8) montre que x est numéroté après u: contradiction, donc $s \in \mathcal{C}_1$.

Soit w le premier sommet numéroté par σ_1. Et soit $\mathcal{C}$ la première clique de la x, y-chaîne de cliques dans $\mathcal{T}$ ($\mathcal{C}$ comprise entre $\mathcal{C}_1$ et $\mathcal{C}_2$). Pour que le dernier sommet numéroté par σ_1 soit u, il faut numéroter des sommets de $N_c(x), N_c(y)$ et $N_c(v)$ avant de numéroter des sommets de $N_c(u)$. Sinon u serait plus proche que l'un des trois sommets x, y, v de l'ensemble des sommets numérotés. Il doit donc exister dans $N_c(w)$ un sommet $w_3 \in N_c(x) \setminus N_c(u)$, $w_4 \in N_c(y) \setminus N_c(u)$ et $w_2 \in N_c(v) \setminus N_c(u)$.

1. Si $w_3 \in N_c(v)$ et $w_4 \in N_c(v)$: nous avons montré l'existence de $a_{13} \in \mathcal{C}_1 \cap \mathcal{C}_2$ (donc dans $\mathcal{C}$) tel que $d(a_{13}, v) = d_2 + 1$, *i.e.* $a_{13} \notin N_c(v)$. Nous savons aussi qu'il existe, dans $\mathcal{C}_2$, un sommet a_{14} tel que $d(a_{14}, v) = d_2 + 1$ et $a_{14} \in N_{c_2}(u) \cap N_{c_2}(y)$.

   Si $a_{14} \in \mathcal{C}$, alors les sommets a_{13}, a_{14}, w_3, w_4 forment la clique d'un 4-soleil. Les sommets du stable sont u', v', x', y' des sommets appartenant à la composante $cc(u)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_u]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant u (*resp.* v, x, y).

   Si $a_{14} \notin \mathcal{C}$, alors les sommets a_{13}, w_3, w_4 forment la clique d'un 3-soleil. Les sommets du stable sont a_{14}, v', x' où v' et x' sont des sommets appartenant à la composante $cc(v)$ (*resp.* $cc(x)$) de $G[V \setminus S_v]$ (*resp.* $G[V \setminus S_x]$) contenant v (*resp.* x).

2. Si un des deux sommets w_3 et w_4 n'appartient pas à $N_c(v)$: par exemple $w_3 \in N_c(v)$ et $w_4 \notin N_c(v)$. Puisque $\varepsilon_{24} = 0$, il existe $a_{24} \in \mathcal{C}_1 \cap \mathcal{C}_2$ (donc $a_{24} \in \mathcal{C}$) tel que $d(a_{24}, v) = d_2$ et $d(a_{24}, y) = d_4$, *i.e.* $a_{24} \in N_c(v) \cap N_c(y)$. Notons que si $a_{24} \in N_c(w)$, nous appliquons le cas précédent, donc $a_{24} \notin N_c(w)$. Puisque $\varepsilon_{34} = 1$, a_{24} et w_3 sont aussi différents. Alors w_3, w_4, a_{24} forment la clique d'un 3-soleil. Les sommets du stable sont w', v', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(y)$) de $G[V \setminus S_w]$ (*resp.* $G[V \setminus S_v]$ $G[V \setminus S_y]$) contenant w (*resp.* v, y).

3. Si $w_3 \notin N_c(v)$ et $w_4 \notin N_c(v)$: dans le cas précédent, nous avons vu que $\varepsilon_{24} = 0$ implique l'existence de $a_{24} \in N_c(v) \cap N_c(y) \setminus N_c(v)$. De plus $\varepsilon_{23} = 0$, donc il existe aussi dans $\mathcal{C}_1$, un sommet $a_{23} \in N_{c_2}(v) \cap N_{c_2}(x)$ tel que $d(a_{23}, y) = d_4 + 1$. Si l'un ou l'autre de a_{23} et a_{24} appartient à $N_c(w)$, nous retrouvons un des deux cas précédents. Supposons donc que $a_{23} \notin N_c(w)$ et $a_{24} \notin N_c(w)$. Enfin $\varepsilon_{34} = 1$ implique que w_3 (*resp.* a_{23}) n'est égal ni à w_4 et a_{24}.

   Si $a_{23} \in \mathcal{C}$, alors les sommets a_{23}, a_{24}, w_3, w_4 forment la clique d'un 4-soleil. Les sommets du stable sont w', v', x', y' des sommets appartenant à la composante $cc(w)$ (*resp.* $cc(v), cc(x), cc(y)$) de $G[V \setminus S_w]$ (*resp.*

$G[V \setminus S_v]$ $G[V \setminus S_x]$, $G[V \setminus S_y]$) contenant w (*resp.* v, x, y).
Si $a_{23} \notin \mathcal{C}$, alors les sommets a_{24}, w_3, w_4 forment la clique d'un 3-soleil. Les sommets du stable sont a_{23}, v', x' où v' et x' sont des sommets appartenant à la composante $cc(v)$ (*resp.* $cc(x)$) de $G[V \setminus S_v]$ (*resp.* $G[V \setminus S_x]$) contenant v (*resp.* x).

□

Nous avons donc montré dans tous les cas de figure l'existence soit d'un 3-soleil, soit d'un 4-soleil. La figure 3.1 montre un exemple de graphe triangulé pour lequel l'heuristique ne trouve pas le sommet d'excentricité maximum.

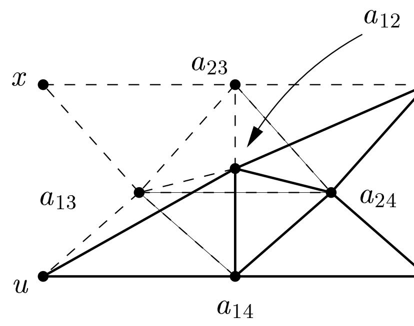


FIG. 3.3 – *Exemple de graphe triangulé où l'heuristique 7 peut échouer. Présence d'un 3-soleil.*

□

Montrons pour terminer le résultat concernant la parité du diamètre d'un graphe triangulé :

Théorème 3.12 *Soit G un graphe triangulé. Si $diam(G)$ est pair, alors l'heuristique 7 termine sur un sommet d'excentricité maximum.*

Preuve : Soit u et v les derniers sommets visités par les deux parcours Lex-BFS de l'heuristique 7. Supposons que leurs excentricités ne soient pas maximum. Si nous reprenons les différentes valeurs de ε_{ij} et des distances obtenues dans les trois cas du théorème précèdent, nous nous rendons compte que $d(u, v)$ est paire et donc le diamètre, égal à $d(u, v) + 1$, est impair. Autrement dit, si l'heuristique 7 échoue alors le diamètre est impair. □

Deuxième partie

Algorithmes de Partitionnement

Chapitre 4

Introduction

Cette seconde partie présente une étude des algorithmes basés sur les *techniques d'affinage de partitions*. Elle a débuté dans l'été 1996 et a été menée, dans sa totalité, avec M. Habib (LIRMM, université de Montpellier) et L. Viennot (LIAFA, université Paris VII), et en partie, avec R. Mc Connell (Université de Denver-Colorado, USA). Les premiers résultats algorithmiques obtenus font l'objet d'un article accepté pour publication dans la revue *Theoretical Computer Science* [HMPV97]. Plus récemment, nous avons élargi notre étude à de nouvelles applications en présentant les différents algorithmes dans un unique formalisme. Une première version de cette extension a été acceptée à la conférence STACS'98 [HPV97]. Nous développons cet aspect tout au long des prochains chapitres.

Nous présentons donc une compilation d'algorithmes, tous basés sur *l'affinage de partition*. Cette technique n'est pas nouvelle. Son application la plus connue est vraisemblablement *"Quicksort"*. Mais elle est apparue aussi dans des domaines très différents. Dès 1971, Hopcroft met au point un algorithme de partitionnement pour la minimisation du nombre d'états d'un automate fini déterministe [Hop71]. Il faut noter que son algorithme est toujours d'actualité. Spinrad [Spi86] s'est intéressé aux processus de partitionnement du point de vue des graphes. Ses travaux (en collaboration avec Mc Connell) ont débouché récemment sur des algorithmes efficaces pour la décomposition modulaire d'un graphe [MS94], l'orientation transitive d'un graphe [MS94, MS97]. Paigue et Tarjan [PT87] ont étudié ce type d'algorithme pour trois applications différentes : le tri lexicographique de chaînes de caractères, le double ordonnancement lexicographique d'une matrice booléenne et une généralisation de la minimisation des automates (calcul de la partition relationnelle maximum). Plus récemment, en collaboration avec Mc Connell, Habib et Viennot [HMPV97], grâce à ces techniques, nous avons simplifié des algorithmes connus pour la reconnaissance de différentes classes de graphes, des matrices possédant la propriété des 1 consécutifs et de généraliser les résultats de Spinrad et Mc Connell sur l'orientation transitive.

L'affinage de partition possède un large champ d'applications : chaînes de caractères, graphes, automates, matrices booléennes. La liste n'est sans doute pas exhaustive. L'utilisation d'une même technique pour des applications différentes n'est sans doute pas le fruit du hasard. Nous nous sommes demandés quels étaient leurs points communs, leurs différences. Pourquoi ne pourrions-nous pas utiliser une astuce algorithmique de l'une de ces applications pour améliorer l'efficacité d'une autre ? Ces applications ne sont-elles pas finalement des traductions différentes d'un même problème ?

Pour répondre à ces questions, le premier travail consiste à proposer un unique formalisme pour implémenter toutes ces applications. C'est ce que nous proposons dans les chapitres qui suivent. A notre connaissance, c'est la première fois qu'une étude globale sur ces applications est menée.

Les premières conséquences de cette formalisation apparaissent dans les preuves de correction et les analyses de complexité des algorithmes. Tous les invariants ont la même forme. Leurs preuves sont très similaires d'une application à l'autre. Leurs simplicités permet une bonne compréhension des points communs entre les applications et de leurs différences.

L'unification de tous ces algorithmes a permis d'en obtenir de nouveaux, notamment pour la décomposition modulaire, l'orientation transitive d'un graphe, la reconnaissance des cographes, la reconnaissance des graphes d'intervalles et le test des 1 consécutifs dans une matrice booléenne. Même si les complexités obtenues ne sont pas systématiquement optimales (certaines applications admettent de meilleurs algorithmes que ceux présentés ici), nous avons souvent simplifié les algorithmes. Il faut noter que les complexités optimales sont éventuellement accrues d'un facteur logarithmique. A la vue des résultats déjà obtenus, il nous semble que ce travail ouvre d'importantes perspectives.

Avant de rentrer plus dans les détails, expliquons brièvement le principe de base de l'affinage de partition. Etant donnée une partition en classes d'un ensemble E, l'idée est de trouver à chaque étape un ensemble pivot S qui permette de couper les classes de la partition en deux (voir l'algorithme ci-dessous).

Algorithme générique d'affinage de partition :

Données : une partition $\mathcal{P} = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ d'un ensemble E en classes

Résultat : un affinage de la partition $\mathcal{P}$

début

  tant que *il existe un ensemble pivot* S **faire**

    pour chaque *classe* $\mathcal{X}$ *de* $\mathcal{P}$ **faire**

      retirer les élements $\mathcal{X}_a$ de $\mathcal{X}$ qui appartiennent à S;

      si $\mathcal{X}$ *est maintenant vide* **alors** remplacer $\mathcal{X}$ par $\mathcal{X}_a$;

      sinon créer la nouvelle classe $\mathcal{X}_a$;

fin

Pour fixer les idées, illustrons notre propos par l'algorithme du *Quicksort*. Soit un ensemble E de n entiers. *Quicksort* trie E par affinage de partition. La partition initiale est constituée de l'unique ensemble E. A chaque étape, un entier pivot p est choisi dans une classe $\mathcal{X}_p$ telle $|\mathcal{X}_p| > 1$. L'élément pivot p est supprimé de $\mathcal{X}_p$. L'ensemble pivot S_p associé est constitué des éléments de $\mathcal{X}_p$ supérieurs à p. Pour respecter l'ordre croissant, la nouvelle classe $X_p \cap S_p = S_p$ est placée après p, et la classe $\mathcal{X}_p \setminus S_p$ avant p. La figure 4.1 illustre ce processus.

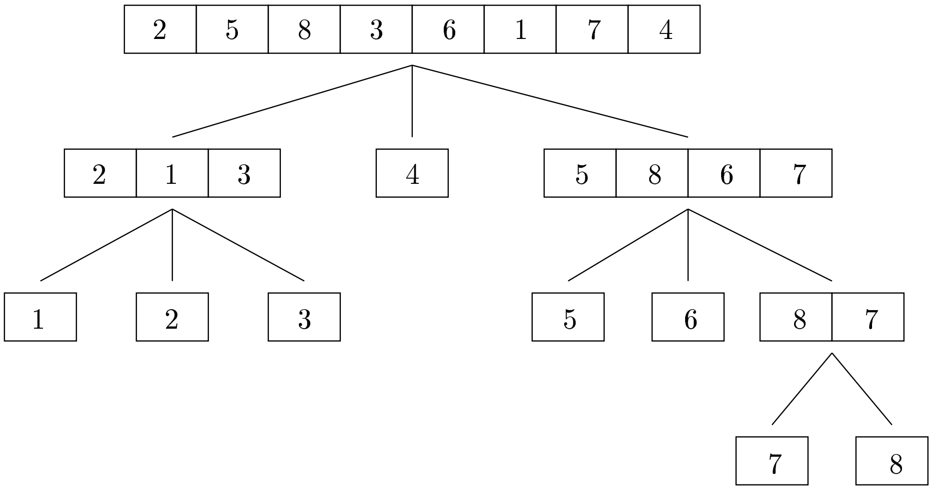


FIG. 4.1 – *Exécution du quicksort*

Dans *Quicksort* seule la classe du pivot est coupée. Dans le cas général, toutes les classes seront susceptibles d'être coupées car S_p ne sera pas nécessairement inclus dans $\mathcal{X}_p$.

Dans *Quicksort* l'ordre sur les classes de la partition est important. Ce ne sera généralement pas le cas. La manière d'insérer les nouvelles classes peut donc varier d'une application à l'autre. Une procédure propre à chaque application tiendra compte des différentes solutions. L'algorithme générique sera aussi paramétré suivant d'autres critères, comme la manière de calculer les ensembles pivots par exemple.

Dans le chapitre 5, nous détaillons la routine principale d'affinage de partition et les différentes procédures paramétrables ainsi que les structures de données utilisées. Afin de se familiariser avec notre formalisme, une première série d'applications simples est présentée au chapitre 6.

Le chapitre 7 abordera des applications plus sophistiquées (partition fonctionnelle maximum, partition relationnelle maximum et minimisation d'automate). Ces algorithmes sont issus de travaux réalisés par Hopcroft [Hop71], Kanellakis et Smolka[KS83] ainsi que par Paigue et Tarjan [PT87]. Ces applications ont comme point commun une même règle de choix des ensembles pivots. Des problèmes d'algorithmiques de graphes utilisant la même règle sont présentés au chapitre 8 : décomposition modulaire, reconnaissance des cographes, orientation transitive. Enfin, le chapitre 9 traite de la reconnaissance des graphes d'intervalles et de la généralisation de ce problème au test des 1 consécutifs d'une matrice booléenne. Pour ces applications nous connaissons une règle simple et efficace et de choix des ensembles pivots.

Chapitre 5

Mise en œuvre

Dans ce chapitre, nous définissons les notions reliées aux partitions qui nous sont utiles. Nous précisons aussi en détails la structure de données que nous utiliserons. Enfin nous détaillons les différentes étapes de l'algorithme générique (voir l'introduction). Pour finir, nous proposons une première classification des différents problèmes abordés.

5.1 Affinage de partitions

Définition 5.1 *Une* ***partition*** $\mathcal{P}$ *d'un ensemble fini* E *est un ensemble de sous-ensembles disjoints* $\{\mathcal{X}_1, \ldots, \mathcal{X}_k\}$*, appelés* classes *de* $\mathcal{P}$ *tels que* $\cup_{1 \leqslant i \leqslant k} \mathcal{X}_i = E$.

L'ordre classique sur les partitions d'un ensemble E est défini ci-dessous. Rappelons qu'il définit le treillis des partitions. Pour Aigner [Aig79], c'est l'un des ordres fondamentaux de la combinatoire.

Définition 5.2 *Soient* $\mathcal{P}$ *et* $\mathcal{Q}$ *deux partitions de* E*. Alors* $\mathcal{P} \preccurlyeq \mathcal{Q}$ *si et seulement si pour toute classe* $\mathcal{X}$ *de* $\mathcal{P}$*, il existe une classe* $\mathcal{Y}$ *de* $\mathcal{Q}$ *telle que* $\mathcal{X} \subseteq \mathcal{Y}$*. Nous dirons que* $\mathcal{P}$ *est* ***plus fine*** *que* $\mathcal{Q}$ *ou que* $\mathcal{Q}$ *est* ***plus grossière*** *que* $\mathcal{P}$.

Définition 5.3 *Nous dirons que* S ***intersecte strictement*** S' *si et seulement si* $S \cap S' \neq \emptyset$ *et* $S \nsubseteq S'$.

Notons que nous pouvons avoir $S' \subseteq S$. L'affinage d'une partition $\mathcal{P}$ de l'ensemble E, suivant un *ensemble pivot* $S \subseteq E$, se définit de la manière suivante :

Définition 5.4 *La partition* $\mathcal{P}' = \mathit{Affiner}(\mathcal{P}, S)$ *est obtenue en remplaçant dans* $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_k\}$ *toutes les classes* $\mathcal{X}_i$*,* $1 \leqslant i \leqslant k$*, qui intersectent strictement* S *par les deux sous-classes* $\mathcal{X}_{i,a} = \mathcal{X}_i \cap S$ *et* $\mathcal{X}_{i,b} = \mathcal{X}_i \setminus S$.

Définition 5.5 *Une partition $\mathcal{P}$ est **stable** pour un ensemble pivot S si aucune classe $\mathcal{X}$ de $\mathcal{P}$ n'intersecte strictement S : i.e. Affiner($\mathcal{P}, S$) $= \mathcal{P}$.*

L'affinage d'une partition $\mathcal{P}$ par un ensemble S produit donc une partition $\mathcal{P}'$ stable pour S et plus fine que $\mathcal{P}$ ($\mathcal{P} \preccurlyeq \mathcal{P}$). Nous dirons aussi qu'un ensemble S *affine strictement* $\mathcal{P}$ si $\mathcal{P}$ n'est pas stable pour S : *i.e.* *Affiner*$(\mathcal{P}, S) \prec \mathcal{P}$.

5.1.1 Structure de données

Présentons les structures de données utilisées pour gérer une partition $\mathcal{P}$ d'un ensemble E. L'objectif est de pouvoir effectuer l'opération *Affiner*$(\mathcal{P}, S)$, où $S \subseteq E$, avec une complexité $O(|S|)$.

Définition 5.6 *Une **partition ordonnée** $\mathcal{P}$ d'un ensemble E est une partition de E munie d'un ordre total sur ses classes.*

Dans l'exemple du *Quicksort*, la partition que nous cherchons est une partition ordonnée (nous trions des éléments). Ce n'est pas le cas pour toutes les applications. Pourtant, cet ordre est utile pour atteindre la complexité souhaitée de l'opération *Affiner*$(\mathcal{P}, S)$. Nous manipulons donc des partitions ordonnées. En effet, la structure de données que nous utilisons est la suivante (voir figure 5.1) :

- Les éléments de E sont contenus dans une liste doublement chaînée $L(E)$. Chaque élément possède un pointeur sur la classe qui le contient.

- Les classes sont des enregistrements indépendants qui sont créés dynamiquement au fur et à mesure du processus d'affinage. Les éléments d'une classe $\mathcal{X}$ sont consécutifs dans la liste $L(E)$, ils forment un intervalle. Chaque classe $\mathcal{X}$ possède un pointeur sur son plus petit et plus grand élément dans $L(E)$.

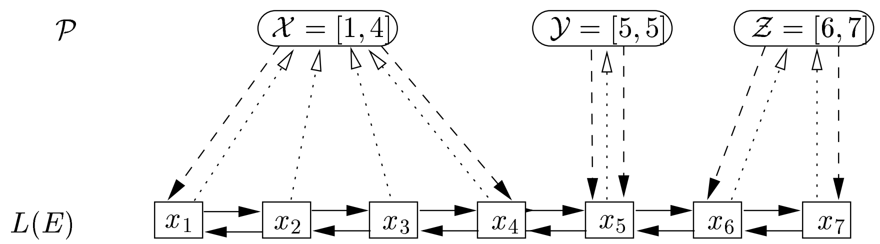


FIG. 5.1 – *Implémentation par listes et pointeurs d'une partition $\mathcal{P}$*

Puisque l'ensemble E est implémenté par une liste, les classes de la partition seront implicitement ordonnées. Etant donné deux éléments, la comparaison de deux pointeurs permet de savoir s'ils appartiennent à la même classe. En parcourant la liste $L(E)$, nous pouvons donc retrouver l'ordre sur les classes.

La consécutivité des éléments d'une classe dans $L(E)$ est primordiale pour la complexité de *Affiner*$(\mathcal{P}, S)$. En ce qui concerne la partition initiale, cette propriété peut facilement être obtenue en temps linéaire par un "tri par dénombrement" [CLR94]. Nous allons voir comment elle peut facilement être maintenue au cours de l'opération *Affiner*$(\mathcal{P}, S)$. Nous dirons qu'une étape d'affinage produit une *partition compatible* avec la partition d'origine :

Définition 5.7 *Une partition ordonnée $\mathcal{Q}$ est* ***compatible*** *avec une partition ordonnée $\mathcal{P}$ si et seulement si $\mathcal{Q} \preccurlyeq \mathcal{P}$ et l'ordre des classes de $\mathcal{Q}$ respecte l'ordre des classes de $\mathcal{P}$ (i.e. si dans $\mathcal{P}$, la classe $\mathcal{X}$ apparaît avant $\mathcal{Y}$, alors les classes de $\mathcal{Q}$ incluses dans $\mathcal{X}$ apparaissent avant celles incluses dans $\mathcal{Y}$).*

Décrivons le principe de *Affiner*$(\mathcal{P}, S)$ (voir figure 5.2). Nous supposons que l'ensemble S possède un pointeur sur chacun de ses éléments dans la liste $L(E)$.

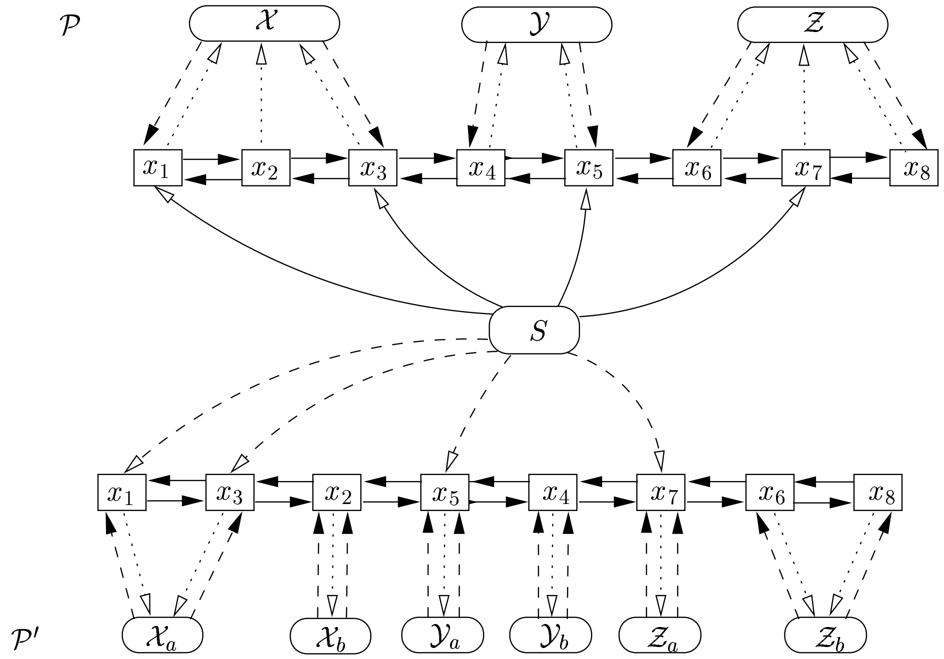


FIG. 5.2 – $\mathcal{P}' =$ *Affiner*$(\mathcal{P}, S)$

Lemme 5.8 *La complexité de l'opération Affiner*$(\mathcal{P}, S)$ *est* $O(|S|)$.

Preuve : Soit $\mathcal{X}$ une classe intersectant strictement S : *i.e.* telle que $\mathcal{X}_a = \mathcal{X} \cap S \neq \emptyset$ et $\mathcal{X}_b = \mathcal{X} \setminus S \neq \emptyset$. Chaque élément de $\mathcal{X}_a$ peut être supprimé de $L(E)$ et ajouté au début de l'intervalle correspondant à la classe $\mathcal{X}$ en temps constant. A chaque déplacement, un compteur est incrémenté de 1. Il permet de savoir si une classe est en cours d'affinage et à la fin de l'affinage, il permet de calculer les nouvelles bornes des classes affinées. Ce compteur doit être remis à 0 à la fin de chaque affinage. A chaque fois qu'une classe est "touchée", nous pouvons ajouter un pointeur sur cette classe à un ensemble $\mathcal{C}$. Il suffit de reparcourir $\mathcal{C}$ pour réinitialiser les compteurs. Seule $\mathcal{X}_a$ est créée, l'ancienne classe $\mathcal{X}$ joue maintenant le rôle de $\mathcal{X}_b$. L'affinage de la classe $\mathcal{X}$ coûte donc $O(|\mathcal{X}_a|)$. Globalement nous obtenons donc une complexité en $O(|S|)$. □

Notons que ,de cette manière, la propriété de consécutivité des éléments d'une classe de $\mathcal{P}$ est bien préservée. Dans certaines applications, la remarque suivante sera importante pour la complexité globale du processus d'affinage.

Remarque 5.9 *Notons que si la classe* $\mathcal{X}$ *est affinée en* $\mathcal{X}_a, \mathcal{X}_b$ *à une étape donnée, alors l'ordre sur les éléments de* $\mathcal{X}_a$ *(resp.* $\mathcal{X}_b$*) respecte l'ordre sur les éléments de* $\mathcal{X}$ *(i.e. soient deux éléments* x *et* y *de* $\mathcal{X}_a$*, alors* $x < y$ *dans* X_a*, si et seulement si* $x < y$ *dans* $\mathcal{X}$*).*

Nous avons décrit l'opération *Affiner*$(\mathcal{P}, S)$ en insérant à droite les éléments de $\mathcal{X}_a$. Il n'y a aucune raison pour que ce soit le cas dans toutes les applications. Nous allons donc paramétrer *Affiner*$(\mathcal{P}, S)$ par une procédure *InsertàDroite* :

- *InsertàDroite* : retourne *vrai* si nous souhaitons insérer $\mathcal{X}_a$ à droite de $\mathcal{X}_b$, *faux* sinon.

Une autre procédure paramètre l'implémentation de *Affiner*$(\mathcal{P}, S)$ (voir procédure 8). Il s'agit de la procédure *AjoutPivot* :

- *AjoutPivot* ajoute le cas échéant, après chaque affinage les nouveaux ensembles pivots dûs aux nouvelles classes et les insère dans une structure appelée *Pivots*.

Prenons l'exemple du *Quicksort* présenté dans l'introduction. Lorsqu'une classe $\mathcal{X}$ est coupée en $\mathcal{X}_a$, $\mathcal{X}_b$, alors il faut choisir un élément dans chacune des deux nouvelles classes. Notons les p_a et p_b. Les ensembles pivots associés sont alors $S_a = \{x \in \mathcal{X}_a : p_a \leqslant x\}$ et $S_b = \{x \in \mathcal{X}_b : p_b \leqslant x\}$. Ils permettront d'affiner $\mathcal{X}_a$ et $\mathcal{X}_b$.

Procédure 8: *Affiner*$(\mathcal{P}, S)$

Données : une partition $\mathcal{P}$ et un ensemble pivot S
Résultat : une partition $\mathcal{P}'$ compatible avec $\mathcal{P}$ et stable pour S
début
  soit M l'ensemble des classes intersectant strictement S (*i.e.* intersectant S et non incluses dans S);
  pour chaque *classe* $\mathcal{X} \in M$ **faire**
    soit $\mathcal{X}_a$ les éléments de $\mathcal{X}$ appartenant à S;
    supprimer $\mathcal{X}_a$ de $\mathcal{X}$;
    si *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$ **alors**
      insérer $\mathcal{X}_a$ à droite de $\mathcal{X}$;
    sinon insérer $\mathcal{X}_a$ à gauche de $\mathcal{X}$;
    AjoutPivot $(\mathcal{X}, \mathcal{X}_a)$;
fin

Nous devons donc stocker ces ensembles pivots pour les utiliser lors de futures étapes. La structure *Pivots* y est dédiée. Nous devons pouvoir ajouter et retirer un élément quelconque de cette structure en temps constant. Pour certaines applications, l'ordre dans lequel ces éléments sont retirés, peut avoir une importance. Pour prendre en compte cette contrainte supplémentaire, *Pivots* peut par exemple être implémenté par une pile ou une file.

En fait, *Pivots* ne contient pas directement les ensembles pivot, mais seulement des pointeurs sur ces ensembles. Cela permet d'utiliser moins de mémoire. Un élément de *Pivots*, noté $\mathcal{E}$, correspond à un sous-ensemble de E. Cela peut être par exemple, une classe de la partition ou le voisinage d'un sommet dans une application sur les graphes. Selon les applications, un élément $\mathcal{E} \in Pivots$ peut générer un ou plusieurs ensembles pivots.

5.1.2 L'algorithme de base

La plupart des algorithmes que nous présentons dans cette partie sont basés sur l'algorithme 9. Il décrit le processus général d'affinage de partition et utilise deux procédures :

- *InitPartition* permet de démarrer le processus d'affinage de la partition $\mathcal{P}$ en proposant une partition initiale et en ajoutant des ensembles pivots à *Pivots*. Elle peut aussi être utilisée pour relancer le processus récursivement à l'intérieur d'une classe lorsque $Pivots$ est vide. Son implémentation dépend de chaque application.

- *ClassePivot* calcule à partir d'un élément de $Pivots$, les ensembles pivots associés et affine la partition avec ces ensembles. Nous détaillons cette procédure après la présentation de l'algorithme général.

Algorithme 9: Affinage de Partition

Données : une partition $\mathcal{P} = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ d'un ensemble E en classes
Résultat : un affinage de la partition $\mathcal{P}$
début
    $Pivots = \emptyset$;
    tant que *la procédure InitPartition n'arrête pas la boucle* **faire**
        InitPartition $(\mathcal{P})$;
        tant que $Pivots \neq \emptyset$ **faire**
            choisir un élément $\mathcal{E}$ de $Pivots$;
            ClassePivot $(\mathcal{E})$;
fin

Intéressons-nous à la procédure *ClassePivot*. La prochaine section présente les différentes applications de l'affinage de partitions abordées dans cette partie. La procédure *ClassePivot* permet déjà d'en distinguer deux qui correspondent chacunes à une implémentation différente. Dans les deux cas, *ClassePivot* fait appel à ces deux autres procédures :

- *EnsemblePivot* calcule l'ensemble pivot S correspondant à un élément $\mathcal{E}$ de $Pivots$ juste avant d'affiner la partition par l'ensemble S. Calculer cet ensemble au dernier moment permet de gagner de l'espace mémoire (seuls des pointeurs sont stockés).
- *Affiner* que nous avons décrite précédemment.

Dans la première catégorie d'applications, il y a autant d'étapes d'affinage que d'éléments dans l'ensemble $\mathcal{E}$ de $Pivots$. Un ensemble pivot par élément de $\mathcal{E}$ est calculé par la procédure *EnsemblePivot*.

La procédure 10 utilise pour le calcul de sommets jumeaux, le tri lexicographique, Lex-BFS, la décomposition modulaire, l'orientation transitive et la reconnaissance des graphes d'intervalles. Voici son schéma :

Procédure 10: *ClassePivot* ($\mathcal{E}$)

Données : un élément $\mathcal{E}$ de $Pivots$
Résultat : Une partition stable pour les ensembles pivots générés par $\mathcal{E}$
début
  pour chaque $x \in \mathcal{E}$ **faire**
    $p \leftarrow (x, \mathcal{E})$;
    $S \leftarrow$*EnsemblePivot* (p);
    affiner$(\mathcal{P}, S)$;
fin

Dans la seconde catégorie, les éléments de l'ensemble $\mathcal{E}$ servent à calculer un seul ensemble pivot. Il n'y a donc qu'une étape d'affinage par classe pivot.

Nous appliquons cette seconde version pour la partition fonctionnelle compatible maximale et pour la partition relationnelle compatible maximale. Nous l'adaptons légèrement pour la minimisation des automates finis déterministes.

Procédure 11: *ClassePivot* ($\mathcal{E}$)

Données : un élément $\mathcal{E}$ de $Pivots$
Résultat : Une partition stable pour l'ensemble pivot généré par $\mathcal{E}$
début
  $S \leftarrow \emptyset$;
  pour chaque $x \in \mathcal{E}$ **faire**
    $p \leftarrow (x, \mathcal{E})$;
    $S \leftarrow S \cup$*EnsemblePivot* (p);
  affiner$(\mathcal{P}, S)$;
fin

Ainsi pour la plupart des applications présentées, il suffit de préciser les petites procédures *InitPartition*, *EnsemblePivot*, *AjoutPivot* et *InsertàDroite*. On peut donc facilement réaliser une bibliothèque d'algorithmes basés sur ce modèle. L. Viennot a commencé à implémenter certaines des applications sous Java.

Afin d'avoir une vue globale du processus d'affinage, nous avons réuni toutes les étapes dans l'algorithme 12. Il faut bien sûr tenir compte des quelques variantes (ici par exemple, pour *ClassePivot*, nous avons choisi la procédure 10)

Algorithme 12: Schéma global de l'affinage de partition

Données : une partition $\mathcal{P} = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ d'un ensemble E en classes
Résultat : un affinage de la partition $\mathcal{P}$

```
début
  Pivots ← ∅;
  tant que la procédure InitPartition n'arrête pas la boucle faire
    InitPartition (P);
    tant que Pivots ≠ ∅ faire
      choisir un élément E de Pivots;
      ClassePivot (E)
        pour chaque x ∈ E faire
          p ← (x, E);
          S ←EnsemblePivot (p);
          affiner(P, S)
            soit M l'ensemble des classes intersectant strictement
            S (i.e. intersectant S et non incluses dans S);
            pour chaque classe X ∈ M faire
              soit X_a les éléments de X appartenant à S;
              supprimer X_a de X;
              si InsertàDroite (X, X_a, p) alors
                  insérer X_a à droite de X;
              sinon insérer X_a à gauche de X;
              AjoutPivot (X, X_a);
fin
```

5.1.3 Invariant

Tous les algorithmes sont prouvés avec un invariant de la forme suivante. Les propriétés A et B sont définies pour chacune des applications :

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Une propriété impliquant l'existence d'une solution compatible avec la partition $\mathcal{P}$*
- ***B:*** *Une propriété indiquant que la partition $\mathcal{P}$ correspond à la solution*

En présentant tous les invariants sous cette forme, nous pouvons aussi utiliser systématiquement une induction sur les étapes d'affinages pour montrer que la propriété A est préservée. Le cas de base est vérifié si la procédure *InitPartition* préserve la propriété A. Dans le cas du *Quicksort*, la propriété A serait la suivante :

– A : Si la classe contenant x est à gauche de la classe contenant y, alors $x < y$.

L'arrêt du processus est caractérisé par la propriété B. Par exemple, pour le *Quicksort* :

– B : Toutes les classes sont des singletons.

Plus généralement, l'arrêt du processus coïncide avec $Pivots = \emptyset$: tous les éléments de $Pivots$ ont été utilisés, donc la partition est stable par les ensembles pivots associés. Nous ne connaissons plus de moyen d'affiner la partition. La procédure *InitPartition* est alors rappelée. Suivant l'application, le processus est soit stoppé, soit relancé en re-partitionnant une ou plusieurs classes “à la main” comme cela a été fait lors du premier appel.

5.1.4 Complexité

L'espace mémoire nécessaire au processus correspond au nombre maximum d'éléments dans $Pivots$. Un élément $\mathcal{E}$ de $Pivots$ n'est pas ajouté une seconde fois. Supposons que $\mathcal{E}$ soit dupliqué. Après que la première copie de $\mathcal{E}$ eut été choisie pour affiner, la partition $\mathcal{P}$ est stable pour les ensembles pivots associés à $\mathcal{E}$. Si nous choisissons une seconde copie de $\mathcal{E}$ dans la foulée, affiner $\mathcal{P}$ à nouveau suivant les ensembles pivots associés à $\mathcal{E}$ ne servirait à rien. Nous ferions alors du travail inutile.

Donc si les éléments de $Pivots$ sont des classes de la partition $\mathcal{P}$ (c'est souvent le cas), à tout instant, la taille de $Pivots$ est bornée par le nombre de classes de la partition. Autrement dit, elle est de l'ordre de $O(n)$ où n est le nombre d'éléments de l'ensemble E.

Chaque ensemble pivot S est calculé au moment de son utilisation pour affiner la partition. La complexité de son calcul est proportionnelle à sa taille $|S|$. Affiner la partition selon S se fait aussi en $O(|S|)$. Pour l'analyse de complexité globale de chaque application, nous utilisons $m = \sum_i |S_i|$, où les ensembles S_i sont les ensembles pivots (nous ne comptons pas les doublons).

La complexité de chaque application dépend uniquement du nombre de fois où un élément de E est utilisé pour calculer un ensemble pivot. Une seule utilisation par élément mène à un algorithme en $O(n + m)$. Pour certaines applications, nous ne savons pas choisir les éléments de E de sorte qu'ils ne soient utilisés qu'une seule fois. Cependant nous verrons une technique générale permettant de borner le nombre d'utilisation de chaque élément par $\log n$, c'est la règle de Hopcroft [Hop71, AHU74]. Les algorithmes sont alors en $O(n + m \log n)$.

5.2 Classifications des différentes applications

Nous avons déjà classé les applications que nous allons étudier en deux catégories selon la procédure *ClassePivot* utilisée. Il existe plusieurs classifications possibles. Il n'est pas possible d'organiser cette partie suivant chacune de ces classifications. Nous avons donc choisi un critère qui nous semble important. Il est toutefois intéressant d'avoir conscience des différentes possibilités, ne serait-ce que pour mieux comprendre les liens entre certaines applications.

5.2.1 Calcul de classes d'équivalences

Dans ce type d'applications, nous calculons une partition de congruence selon une relation d'équivalence donnée. L'ordre sur les classes de la partition n'a donc aucune importance.

Pour ce type de problèmes, l'affinage de partition peut être intéressant si l'information, "x et y n'appartiennent pas à la même classe d'équivalence", est facile à calculer. Il faut ici faire le parallèle avec les techniques "d'union-find" [Tar75] qui s'appliquent lorsque l'information inverse (*i.e.* "x et y appartiennent à la même classe d'équivalence") est facilement accessible.

Les applications que nous étudions dans cette catégorie sont : le calcul de sommets jumeaux, minimisation d'automates (calcul des classes d'équivalences de Nérode), les partitions compatibles maximales et la décomposition modulaire.

Nous pouvons distinguer deux types d'applications ici : celles où le résultat souhaité est la partition de congruence maximale (*exemples :* calcul de sommets jumeaux, la minimisation d'automates et partitions compatibles maximales) et celles où nous relançons le processus récursivement à l'intérieur de chaque classe d'équivalence (*exemple* : décomposition modulaire). Dans les premières, un seul appel à la procédure *InitPartition* est nécessaire, le cardinal des classes obtenues est variable. Dans les secondes, la procédure est appelée jusqu'à ce que toutes les classes soient des singletons.

5.2.2 Partitions ordonnées

Ici, le résultat cherché est en fait un ordre total sur les éléments de E. Autrement dit, nous cherchons une permutation des éléments de E vérifiant une certaine propriété. Il faut donc relancer le processus avec la procédure *InitPartition* autant de fois que nécessaire. Le processus s'arrête lorsque toutes les classes sont des singletons.

Pour Lex-BFS, la reconnaissance des cographes, l'orientation transitive, la reconnaissance des graphes d'intervalles et le test des 1 consécutifs d'une matrice booléenne, nous calculons une partition ordonnée.

5.2.3 Règle de choix du pivot

Nous avons décidé d'organiser cette partie suivant la règle de choix du pivot ("tie-break"). Ce critère est très important car il détermine l'efficacité de l'algorithme. Il y a trois grandes catégories :

1. *Règle simple :* Nous pouvons distinguer deux types de stratégies. Soit nous connaissons à l'avance les pivots que nous allons utiliser (*exemple* calcul des sommets jumeaux, tri lexicographique). Il n'y a alors pas besoin de règle de choix. Soit nous disposons d'une règle très simple : choix arbitraire d'un élément dans une classe (*exemple* Lex-BFS). Nous présentons ces algorithmes dans le chapitre 6. Ils ont tous une complexité linéaire.

2. *Règle de Hopcroft :* Nous avons vu dans la description générale du processus que l'affinage d'une classe est susceptible de créer de nouveaux pivots qui affinent strictement la partition. Cependant ces "bons" pivots ne sont pas toujours faciles à détecter. Hopcroft a mis au point la règle de la *"plus petite moitié"* pour la minimisation du nombre d'états d'un automate fini déterministe [Hop71].

   L'idée est d'insérer dans $Pivots$ la plus petite des deux sous-classes créées. Nous assurons ainsi que chaque élément de E est utilisé au plus $\log n$ fois (où $n = |E|$) pour le calcul des ensembles pivots. Cette règle intervient aussi dans le calcul de la partition fonctionnelle [AHU74] ou relationnelle compatible maximale [PT87], pour la décomposition modulaire [MS94, MS97], pour l'orientation transitive d'un graphe [MS94, MS97, HMPV97] et pour la reconnaissance des graphes d'intervalles [HMPV97, HPV97].

3. *Règle astucieuse :* Il s'agit de trouver une méthode garantissant que chaque élément de E est utilisé une seule fois pour pivoter. Mc Connell et Spin-

rad [MS94, MS97] ont montré que pour l'orientation transitive, il était possible de choisir les pivots de manière à n'utiliser chaque sommet du graphe qu'une seule fois comme pivot. Ces deux auteurs, ainsi que Habib et Cournier [CH94], ont mis en évidence le même type de résultat pour la décomposition modulaire. Ils utilisent pour cela des techniques très compliquées. A notre connaissance, la seule application, admettant une règle de choix de pivots non-triviale qui soit simple à implémenter, est la reconnaissance des graphes d'intervalles [HMPV97]. Nous présentons cette application ainsi que sa généralisation au test des 1 consécutifs d'une matrice booléenne.

Chapitre 6

Règle élémentaire de choix des pivots

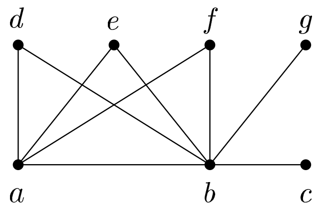
Dans ce chapitre, nous présentons des applications très simples : calcul des sommets jumeaux d'un graphe simple, tri lexicographique de chaînes de caractères et l'algorithme Lex-BFS. Certaines peuvent être considérées comme des exercices d'algorithmique de second cycle. Dans tous les cas, le système de sélection des pivots est très simple. Soit les éléments de $Pivots$ sont tous connus à l'avance, soit ils sont choisis arbitrairement parmi les éléments d'une classe. Ainsi l'analyse de complexité reste facile. Les preuves de correction sont souvent triviales. Nous ne les faisons pas, le lecteur intéressé se référera aux articles cités.

6.1 Calcul des sommets jumeaux

Nous présentons dans cette section deux applications : le calcul des sommets jumeaux d'un graphe simple et le calcul d'une extension linéaire de l'ordre d'inclusion des voisinages des sommets d'un graphe. Nous commencons par présenter la première citée car elle nous semble la plus classique. Puis nous verrons que l'algorithme présenté permet aussi de résoudre le second problème.

Dans un graphe, nous notons $N(u)$ le *voisinage ouvert* du sommet u (*i.e.* le voisinage au sens classique) et $N[u] = N(u) \cup \{u\}$ son *voisinage clos*. Nous allons chercher quels sont les sommets d'un graphe ayant le même voisinage ouvert ou fermé.

Définition 6.1 *Soient u et v deux sommets d'un graphe G. On dit que u et v sont des* vrais jumeaux *si $N(u) = N(v)$, et des faux jumeaux si $N[u] = N[v]$.*



$$\mathcal{P} = \{\{a,b,c,d,e,f,g\}\}$$

pivot $= a$ $\quad \mathcal{P} = \{\{a,c,g\},\{b,d,e,f\}\}$

pivot $= c$ $\quad \mathcal{P} = \{\{a,c,g\},\{d,e,f\},\{b\}\}$

pivot $= e$ $\quad \mathcal{P} = \{\{c,g\},\{a\},\{d,e,f\},\{b\}\}$

FIG. 6.1 – *Un graphe et la partition des sommets en classe de vrais jumeaux. En affinant selon $N(a), N(c)$ et $N(e)$, nous obtenons une partition stable.*

Intéressons-nous dans un premier temps au calcul des vrais jumeaux. Nous cherchons donc une partition des sommets d'un graphe G telle que deux sommets appartiennent à la même classe si et seulement si ce sont de vrais jumeaux. La relation binaire "être vrais jumeaux" est clairement réflexive, symétrique et transitive. Nous calculons donc des classes d'équivalence d'une relation binaire. L'ordre sur les classes de la partition n'a aucune importance.

Algorithme 13: Calcul des sommets jumeaux

Données : Un graphe $G = (V, E)$ sous forme de listes d'adjacence

Résultat : Une partition $\mathcal{P} = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ des sommets de G en classes de vrais jumeaux

début

- $\mathcal{P} \leftarrow \{V\}$;
- exécuter l'algorithme 9 avec la procédure 10 et les définitions suivantes (l'unique élément de *Pivots* est V);

fin

Procedure *InitPartition* ($\mathcal{P}$)

- *au premier appel:*
  - ajouter V à *Pivots*;
- *au second appel:*
  - fin du processus, **retourner** la partition $\mathcal{P}$;

Procedure *EnsemblePivot* ($p = (x, \mathcal{X})$)

- **retourner** la liste d'adjacence du sommet x;

Procedure *InsertàDroite* ($\mathcal{X}, \mathcal{X}_a, p$)

- **retourner** vrai;

Procedure *AjoutPivot* ($\mathcal{X}, \mathcal{X}_a$)

- rien;

Notons que les ensembles pivots sont connus à l'avance. Donc la procédure *AjoutPivot* ne fait rien et la procédure *InitPartition* n'est appelée que deux fois. Il n'y a donc pas besoin de règle de choix pour relancer le processus.

La correction de l'algorithme 13 peut facilement être montrée à l'aide de l'invariant suivant :

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Si deux sommets u et v sont vrais jumeaux, alors ils appartiennent à la même classe.*
- ***B:*** *La partition $\mathcal{P}$ est stable pour tout voisinage ouvert $N(v)$ où v est un sommet de G.*

Preuve : Si u et v sont vrais jumeaux, alors aucun voisinage ouvert $N(x)$ ne peut les séparer. La propriété A est donc toujours vérifiée. Enfin si la propriété B n'est pas vérifiée, alors il existe u et v, deux sommets appartenant à une même classe que $N(x)$ sépare : u et v ne sont pas vrais jumeaux. □

Analyse de complexité : L'algorithme 13 est linéaire. Chaque étape d'affinage coûte $O(|N(x)|)$ où x est le sommet pivot. Puisque chaque sommet est utilisé exactement une fois, nous obtenons une complexité en temps $O(n+m)$ (m le nombre d'arêtes de G). □

Enfin, pour adapter l'algorithme au calcul des faux jumeaux, il suffit que la procédure *EnsemblePivot* retourne le voisinage clos de x, $N[x]$.

Remarque 6.2 *Si u et v sont deux vrais jumeaux, alors il n'existe pas de sommet w tel que u et w soient faux jumeaux. Si c'était possible, u et w seraient adjacents. Donc v et w le seraient aussi. Puisque u et v sont vrais jumeaux, ils sont non-adjacents. Dans ce cas $N[w]$ n'est pas égal à $N[u]$.*

Cette remarque montre que les classes de faux jumeaux et les classes de vrais jumeaux sont disjointes. Autrement dit, les faux jumeaux apparaissent dans des classes singletons de la partition en vrais jumeaux. Nous pourrions donc chercher une partition des sommets de G en classes de vrais ou faux jumeaux. Pour cela, une solution est de faire le calcul en deux passes : une pour calculer les classes de faux jumeaux et l'autre pour calculer les classes de vrais jumeaux. Il suffit alors de réunir dans une unique partition les classes qui ne sont pas des singletons.

Ordre d'inclusion des voisinages des sommets d'un graphe

Nous avons vu que la partition $\mathcal{P}$ est implémentée par une liste chaînée sur l'ensemble E et par des pointeurs entre les éléments de E et leur classe. Si nous considérons le résultat de l'algorithme 13 comme une partition ordonnée, nous obtenons une *extension linéaire de l'ordre d'inclusion des voisinages* des sommets de $G = (V, E)$. Expliquons-nous plus en détails.

Une extension linéaire L d'un ordre partiel P est un ordre total respectant les comparabilités de P : *i.e.* si $x <_P y$, alors $x <_L y$.

Lemme 6.3 *L'algorithme 13 calcule un ordre total sur les sommets correspondant à une extension linéaire de l'ordre d'inclusion de leur voisinage.*

Preuve : L'idée est très simple. Soient deux sommets x et y tels $N(x) \subsetneq N(y)$, alors $N(y) \setminus N(x) \neq \emptyset$. Considérons le premier sommet $z \in N(y) \setminus N(x)$ utilisé comme pivot dans l'algorithme 13. Avant l'affinage de $\mathcal{P}$ par $N(z)$, les sommets x et y appartiennent à la même classe. Cette étape d'affinage les sépare et place x dans une classe à gauche de la classe contenant y. Donc l'ordre total calculé respecte bien les comparabilités de l'ordre d'inclusion des voisinages. C'est une extension linéaire de cet ordre. $\square$

Nous verrons en quoi l'application suivante peut-être considérée comme une généralisation de ce problème.

6.2 Tri lexicographique de chaînes de caractères

L'ordre lexicographique sur des chaînes de caractères n'est rien d'autre que l'ordre usuel des mots d'un dictionnaire. Définissons-le plus formellement.

Soit Σ un alphabet ordonné de k symboles. L'ensemble Σ^* dénote l'ensemble des mots de longueur finie sur Σ. Pour chaque mot x de Σ^*, la i-ème lettre de i est notée $x(i)$. Le mot x est plus petit dans l'ordre lexicographique que le mot y ($x <_{lex} y$) si et seulement si :

- soit il existe j tel que $x(j) < y(j)$ et pour tout i, $1 \leqslant i < j$, $x(i) = y(i)$
- soit x est un préfixe strict de y

Aho, Hopcroft et Ullman [AHU74] ont considéré ce problème comme une généralisation du tri par compartiment (*"bucket sort"*). Ils ont proposé un algorithme en $O(m+k)$ où m est la longueur de toutes les chaînes concaténées. L'algorithme 14 est une implémentation de leur algorithme.

Par nature, la partition que nous cherchons est une partition ordonnée. Ici les pivots sont des ensembles de chaînes de caractères : il faut séparer les mots ayant

Algorithme 14: Tri lexicographique de chaînes de caractères

Données : n chaînes de caractères $x_1, \ldots, x_n$ sur l'alphabet Σ

Résultat : une partition $\mathcal{P} = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ des chaînes de caractères triée dans l'ordre lexicographique

début

  $\mathcal{P} \leftarrow (\{x_1, \ldots, x_n\})$;

  exécuter l'algorithme 9 avec la procédure 10 et les définitions suivantes (les éléments de $Pivots$ sont des alphabets $\Sigma' \subseteq \Sigma$);

fin

Procedure *InitPartition* $(\mathcal{P})$

  au i-ème appel (i < l la longueur du plus grand mot) :

    calculer les ensembles non-vides $S_{i,a}$ de chaînes ayant la lettre a à la position i;

    ajouter à $Pivots$, $\Sigma' = \{a \in \Sigma \cup \{\varepsilon\} : S_{i,a} \neq \emptyset\}$ de telle manière que les lettres soient choisies dans l'ordre alphabétique;

  au l-ème appel :

    fin du processus, **retourner** $\mathcal{P}$;

Procedure *EnsemblePivot* $(p = (a, \Sigma'))$

  retourner $S_{i,a}$;

Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$

  retourner vrai;

Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_a)$

  rien

la lettre α à la position i des autres. Pour minimiser l'espace mémoire, l'unique élément de $Pivots$ est à chaque fois l'ensemble des lettres $a \in \Sigma \cup \{\varepsilon\}$ pour lesquelles $S_{i,a} \neq \emptyset$. La taille de $Pivots$ est donc en $O(k)$.

Comme pour le calcul des jumeaux, nous connaissons à l'avance l'ensemble des pivots : les ensembles $S_{i,a} \neq \emptyset$ pour $1 \leqslant i < l$ et tout $a \in \Sigma \cup \{\varepsilon\}$. Nous aurions donc pu les ajouter tous à $Pivots$ dès le premier appel à la procédure *InitPartition*. Nous avons choisi la solution d'appeler *InitPartition* une fois par indice i pour présenter une légère amélioration de la complexité due à Paigue et Tarjan.

Les pivots étant connus à l'avance, il n'y a pas de règle de choix. L'invariant de l'algorithme 14 est le suivant. La preuve est laissée au lecteur (pour plus de détails voir [AHU74, PT87]) :

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Un tri lexicographique des chaînes de caractères est compatible avec $\mathcal{P}$.*
- ***B:*** *Deux chaînes de caractères différentes n'apparaissent pas dans la même classe.*

Analyse de complexité : Le calcul des ensembles pivots $S_{i,a}$ coûte $O(m+k)$ (n est négligeable devant m). L'ajout des ensembles $S_{i,a}$ à *Pivots* dans l'ordre adéquat peut se faire par un "bucket sort". La complexité de l'ensemble de ces tris est $O(m)$. Puisque chaque ensemble pivot n'est utilisé qu'une fois, nous obtenons une complexité globale en $O(m+k)$. □

Paigue et Tarjan [PT87] ont amélioré cet algorithme grâce à un pré-traitement sur les chaînes de caractères. L'idée est qu'il n'est pas nécessaire de parcourir chaque chaîne entièrement :

Définition 6.4 *Nous appelons* préfixe propre *d'une chaîne de caractères x, le plus petit préfixe qui distingue x des autres chaînes dans l'ordre lexicographique.*

Trier un ensemble de chaînes de caractères dans l'ordre lexicographique est équivalent à trier l'ensemble de leurs préfixes propres. Soit m' la longueur de la concaténation de tous les préfixes propres d'un ensemble E de chaînes de caractères. Trier E peut se faire en $O(m'+k)$. Dans certains cas, cette amélioration est significative.

Les préfixes propres peuvent être calculés au cours de l'exécution de l'algorithme 14. Il faut pouvoir détecter qu'une classe donnée ne sera plus jamais affinée. Cette situation se produit lorsque :

– une classe ne contient plus qu'un seul élément.
– toutes les chaînes de caractères d'une classe sont de longueur i, où i correspond au nombre de fois que la procédure *InitPartition* a déjà été appelée.

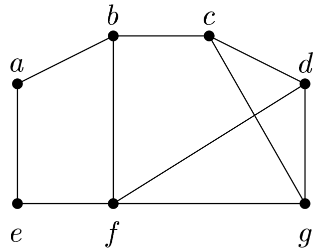
Il reste donc à intégrer ces tests dans l'algorithme 14. L'objectif de ce chapitre étant de présenter des applications simples pour familiariser le lecteur à l'affinage de processus, nous ne présentons pas cette amélioration ici. Le lecteur intéressé peut se référer à [PT87].

Remarque 6.5 *Revenons un instant sur le calcul d'une extension linéaire de l'ordre d'inclusion des voisinages d'un graphe. Considérons l'alphabet comme l'ensemble des sommets du graphe et ordonnons cet alphabet suivant l'ordre de choix* *des pivots. On peut considérer les voisinages comme des chaînes de caractères. Alors calculer une extension linéaire de leur ordre d'inclusion revient à les trier lexicographiquement. Il y a deux différences importantes. La première est que les voisinages sont tous triés suivant l'ordre de l'alphabet (ce n'est clairement pas le cas pour les chaînes de caractères quelconques). La seconde est que le graphe est simple et donc les voisinages ne contiennent pas deux fois la même lettre. De ce point de vue, le tri lexicographique est une généralisation du calcul de l'ordre d'inclusion des voisinages d'un graphe.*

6.3 Lex-BFS

Comme nous l'avons déjà vu au chapitre 1, Lex-BFS est un parcours en largeur qui fait appel à une règle de choix des sommets basée sur l'ordre Lexicographique. Cet algorithme, dû à Rose, Tarjan et Lueker [RTL76] servait initialement à reconnaître les graphes triangulés. Depuis, de nouvelles applications à Lex-BFS ont été découvertes. Nous avons vu par exemple au chapitre 2 qu'il joue un rôle important dans l'approximation du diamètre pour un certain nombre de classes de graphes. Nous verrons aussi son application dans la reconnaissance des graphes d'intervalles.

Dans l'article d'origine [RTL76], l'implémentation de Lex-BFS avait été présentée comme un affinage de partition. Dans le chapitre 1, nous l'avons présenté sous la même forme que Golumbic l'avait fait dans son livre [Gol80]. Dans cette présentation, sans doute la plus connue, les étiquettes associées aux sommets sont explicitement gérées. L'aspect affinage de partition a été oublié. Nous proposons ici une implémentation basée sur la procédure 9. Il n'est pas nécessaire de gérer les étiquettes, elles sont implicites dans l'ordre des classes. De ce point de vue, l'algorithme est simplifié. Un exemple d'exécution de cette implémentation est donnée à la figure 6.2.



$$\mathcal{P} = \{\{a, b, c, d, e, f, g\}\}$$
$$\mathcal{P} = \{\{c, d, f, g\}, \{b, e\}, \{a\}\}$$
$$\mathcal{P} = \{\{d, g\}, \{c, f\}, \{e\}, \{b\}, \{a\}\}$$
$$\mathcal{P} = \{\{d, g\}, \{c\}, \{f\}, \{e\}, \{b\}, \{a\}\}$$
$$\mathcal{P} = \{\{g\}, \{d\}, \{c\}, \{f\}, \{e\}, \{b\}, \{a\}\}$$

FIG. 6.2 – *Exécution de Lex-BFS. L'ordre sur les pivots est* a, b, e, f, c, d, g.

Rappelons la définition d'un ordre lexicographique sur les sommets d'un graphe. Etant donné une numérotation $\sigma = [x_1, \ldots, x_n]$ des sommets, $N_d(x_i)$ dénote le voisinage à droite de x_i, *i.e.* $N_d(x_i) = N(x_i) \cap \{x_{i+1}, \ldots, x_n\}$. Un *ordre lexicographique* sur les sommets d'un graphe est une numérotation des sommets telle que $N_d(x) <_{lex} N_d(y)$ implique $x < y$.

Par nature, la partition que nous cherchons est ordonnée. Les pivots ne sont pas connus à l'avance. Cependant la règle d'ajout des pivots est très simple : choix arbitraire d'un sommet dans une classe. Il faut noter qu'elle est prise en charge par la procédure *InitPartition*. Nous pouvons donc aussi considérer qu'il ne s'agit pas réellement d'un ajout de pivot mais plutôt d'une relance du processus de partitionnement.

Algorithme 15: Lex-BFS

Données : Un graphe $G = (V, E)$ sous forme de listes d'adjacence
Résultat : Un ordre Lex-BFS des sommets de G dans la liste $\mathcal{P}$
début
  $\mathcal{P} \leftarrow (V)$;
  exécuter l'algorithme 9 avec la procédure 10 et les définitions suivantes (les éléments de *Pivots* sont des classes de $\mathcal{P}$);
fin
Procedure *InitPartition* $(\mathcal{P})$
  au i-ème appel :
    si $i = n$ **alors** fin du processus, **retourner** $\mathcal{P}$;
    choisir un sommet x dans la i-ème classe $\mathcal{X}$ en partant de la droite;
    si $|\mathcal{X}| \neq 1$ **alors** remplacer $\mathcal{X}$ par $\mathcal{X} \setminus \{x\}, \{x\}$ dans $\mathcal{P}$;
    ajouter $\{x\}$ à *Pivots*;
Procedure *EnsemblePivot* $(p = (x, \mathcal{X}))$
  retourner la liste d'adjacence du sommet x;
Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$
  retourner vrai;
Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_a)$
  rien

L'invariant de l'algorithme 15 est présenté ci-dessous. La preuve est directe, nous ne la ferons pas (le lecteur intéressé peut se référer à l'article d'origine [RTL76] ou à [Gol80]).

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Il existe un ordre lexicographique des sommets de G compatible avec $\mathcal{P}$.*
- ***B:*** *Toutes les classes sont des singletons.*

Analyse de complexité : L'algorithme 15 est linéaire en temps et en espace. En effet, *Pivots* contient au plus un élément. A la i-ème étape, on ajoute dans *Pivots* un sommet de la i-ème classe. Ce sommet est inséré comme une classe à droite de la i-ème classe. Chaque sommet ne peut donc être utilisé qu'une fois. Chaque étape d'affinage coûte $O(|N(p)|)$ où p est le sommet pivot. Nous obtenons une complexité en temps $O(n+m)$ (n le nombre de sommets et m le nombre d'arêtes de G). □

6.4 Conclusion

Ces trois applications nous ont permis de nous familiariser avec les processus d'affinage de partition. La simplicité de ces exemples provient de la règle d'ajout de pivots. Soit nous connaissons les pivots à l'avance, nous pouvons alors considérer la règle comme inexistante. Soit il s'agit d'un choix arbitraire de l'élément pivot dans une classe (Lex-BFS). Nous avons alors remarqué qu'il s'agit plus d'une relance du processus d'affinage que d'un ajout de pivot.

Dans l'implémentation, cela se traduit par trois procédures *AjoutPivot* qui ne font rien. A chaque fois, nous obtenons un algorithme linéaire. Nous verrons que si *AjoutPivot* contient des instructions, alors la complexité linéaire peut être remise en question.

Toutefois ces trois applications sont simples. Nous présentons dans la suite des problèmes plus complexes où il est nécessaire de gérer l'ajout de pivots. Nous montrons comment une règle simple mise au point par Hopcroft [Hop71], assure de bonnes complexités.

Chapitre 7

Règle de Hopcroft

Contrairement au chapitre 6, les applications présentées ici ne possèdent pas de règle simple de choix d'ajout des pivots. Pour assurer une complexité en $O(m \log n)$ (où n est le nombre d'éléments de l'ensemble à partitionner et m la somme des tailles des ensembles pivots), nous devons utiliser une règle due à Hopcroft [Hop71, AHU74], dite *"règle de la plus petite moitié"* : lorsque nous avons le choix d'affiner la partition courante avec plusieurs classes, nous utilisons une classe dont le cardinale est inférieur à la moitié du cardinal de l'union de ces classes.

Dans un premier temps, nous présentons cette règle appliquée au calcul d'une partition fonctionnelle maximum. Ce problème a été étudié par Hopcroft [Hop71, AHU74]. Nous pourrons mettre en évidence des conditions suffisantes à l'application de cette règle. Puis nous abordons la généralisation de ce problème, due à Hopcroft, à la minimisation d'automates finis, déterministes et complets [Hop71, BBC92]. Enfin nous considérons une seconde généralisation issue des systèmes de calculs communiquant (CCS) : la partition relationnelle maximum. Ce problème a été présenté par Kanellakis et Smolka [KS83]. Le meilleur algorithme est dû à Paigue et Tarjan [PT87]. Cette application nous montrera d'autres conditions autorisant l'application de la règle de Hopcroft.

7.1 Partition fonctionnelle maximum

Le problème que nous présentons ici a été étudié par Hopcroft [Hop71, AHU74] qui l'a ensuite généralisé à la minimisation d'automates finis déterministes. Nous traiterons ce second problème à la section 7.2. Même si l'application aux automates est relativement directe, il est toutefois intéressant d'introduire ce problème "intermédiaire" pour une meilleure compréhension de l'algorithme de minimisation des automates d'une part et pour une future généralisation d'autre part : partition relationnelle maximum (voir section 7.3). Enfin, il permet d'expliquer clairement *"la règle de la plus petite moitié"*.

Définition 7.1 *Soient une partition $\mathcal{Q} = \{\mathcal{B}_1, \ldots, \mathcal{B}_p\}$ d'un ensemble E à n éléments et une fonction $f : E \to E$. Une **partition fonctionnelle** pour la partition $\mathcal{Q}$ et pour la fonction f est une partition $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_q\}$ de E telle que :*

1. *$\mathcal{P}$ est* plus fine *que $\mathcal{Q}$: i.e. $\mathcal{P} \preccurlyeq \mathcal{Q}$.*
2. *$\mathcal{P}$ est* stable *pour f : i.e. si x et y appartiennent à $\mathcal{X}_i \in \mathcal{P}$, alors $f(x)$ et $f(y)$ appartiennent à une même classe $\mathcal{X}_j \in \mathcal{P}$.*

Remarque 7.2 *La partition que nous cherchons n'est pas pas ordonnée. Cependant pour obtenir la complexité voulue, la structure de données que nous utilisons gère une partition ordonnée. Donc la partition $\mathcal{P}$ est compatible avec les partitions courantes que nous calculons.*

La partition fonctionnelle qui nous intéresse est celle ayant le moins de classes. Nous montrerons plus tard que :

Lemme 7.3 *Il existe une unique partition fonctionnelle maximum de $\mathcal{Q}$.*

Dans la suite, la partition $\mathcal{P}$ dénote la partition courante du processus d'affinage. Elle est initialisée avec $\mathcal{Q}$. A chaque étape d'affinage, les ensembles pivots seront l'inverse d'une classe $\mathcal{X}$ de $\mathcal{P}$ (*i.e.* union des images inverses de ses éléments), notée $f^{-1}(\mathcal{X})$. Lorsqu'il n'y a pas d'ambiguïté, nous disons abusivement que $\mathcal{P}$ est stable pour $\mathcal{X}$, si elle est stable pour $f^{-1}(\mathcal{X})$.

L'idée intéressante d'Hopcroft pour résoudre le problème de la partition fonctionnelle maximum repose sur la propriété suivante :

Lemme 7.4 *Soit l'ensemble $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$, l'union de classes de la partition $\mathcal{P}$ tel que $\mathcal{P}$ soit stable pour $f^{-1}(\mathcal{C})$. Alors Affiner($\mathcal{P}, f^{-1}(\mathcal{X}_j)$), $i \leqslant j \leqslant k$, produit la même partition que Affiner($\mathcal{P}, f^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$)*

Preuve : Supposons que *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{X}_j))$ affine $\mathcal{Y}$ en $\mathcal{Y}_a = \mathcal{Y} \cap f^{-1}(\mathcal{X}_j)$ et $\mathcal{Y}_b = \mathcal{Y} \setminus f^{-1}(\mathcal{X}_j)$; et que *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{C} \setminus \mathcal{X}_j))$ affine $\mathcal{Y}$ en $\mathcal{Y}_1 = \mathcal{Y} \cap f^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ et $\mathcal{Y}_2 = \mathcal{Y} \setminus f^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$.

Les seules classes de $\mathcal{P}$ susceptibles d'être affinées par $f^{-1}(\mathcal{X}_j)$ ou $f^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ sont les classes $\mathcal{Y}$ telles que $\mathcal{Y} \subseteq f^{-1}(\mathcal{C})$ (*i.e.* $\forall y \in Y,\ f(y) \in E$) car $\mathcal{P}$ est stable pour $f^{-1}(\mathcal{C})$. Soit $\mathcal{Y}$ une telle classe.

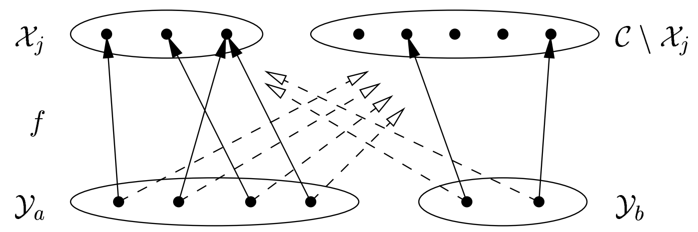


FIG. 7.1 – *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{X}_j)) =$ *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{C} \setminus \mathcal{X}_j))$ *(les flèches en pointillés indiquent qu'un élément n'a pas d'image par f dans un ensemble)*

Puisque tout élément de E possède au plus une image par f, et que $\mathcal{Y} \subseteq f^{-1}(\mathcal{C})$, $f^{-1}(E) = f^{-1}(\mathcal{X}_j) \cup f^{-1}(\mathcal{C} \setminus \mathcal{X}_i)$ (l'union est disjointe). Donc $\mathcal{Y} = (\mathcal{Y} \cap f^{-1}(\mathcal{X}_j)) \cup (\mathcal{Y} \cap f^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$. Autrement dit $\mathcal{Y} = \mathcal{Y}_a \cup \mathcal{Y}_1$ (c'est toujours une union disjointe). Donc $\mathcal{Y}_1 = \mathcal{Y}_b$ et $\mathcal{Y}_a = \mathcal{Y}_2$. Cela signifie que *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{X}_j)) =$ *Affiner*$(\mathcal{P}, f^{-1}(\mathcal{C} \setminus \mathcal{X}_j))$.

Notons que la propriété est en fait vraie pour tout sous-ensemble $\mathcal{C}'$ de $\mathcal{C}$. Il n'est pas nécessaire que $\mathcal{C}'$ soit une classe de la partition. L'algorithme utilise une classe. □

Si la partition courante $\mathcal{P}$ a été affinée par l'ensemble pivot $S = f^{-1}(\mathcal{X})$ où $\mathcal{X}$ est une classe de $\mathcal{P}$, ce n'est plus la peine d'utiliser les éléments de $\mathcal{X}$ pour calculer un ensemble pivot tant que $\mathcal{X}$ n'a pas été elle-même affinée. Nous pouvons dès maintenant présenter la règle de Hopcroft :

Règle de Hopcroft (règle de la plus petite moitié) : Soit l'ensemble $C = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$, union de classes de la partition courante $\mathcal{P}$, tel que $\mathcal{P}$ soit stable pour $f^{-1}(\mathcal{C})$. Alors nous choisissons d'affiner $\mathcal{P}$, avec une classe $\mathcal{X}_j$, $i \leqslant j \leqslant k$, telle que $|X_j| \leqslant |\mathcal{X}_i \cup \ldots \cup \mathcal{X}_k|/2$.

Remarque 7.5 (Commutativité) *Si la partition $\mathcal{P}$ n'est pas stable pour $f^{-1}(\mathcal{X})$ et $f^{-1}(\mathcal{Y})$, alors :*

Affiner(*Affiner*$(\mathcal{P}, f^{-1}(\mathcal{X})), f^{-1}(\mathcal{Y})) =$ *Affiner*(*Affiner*$(\mathcal{P}, f^{-1}(\mathcal{Y})), f^{-1}(\mathcal{X}))$

Cette propriété découle de l'utilisation d'opérations ensemblistes (intersection et différence). Elle n'est pas nécessaire à la preuve de l'algorithme.

La figure 7.2 illustre le principe de l'algorithme. Expliquons la mise à jour de *Pivots*. Il ne contiens que les classes pour lesquelles la partition n'est nécessairement stable. Il est donc initialisé avec toutes les classes de la partition initiale.

Lorsque la partition courante $\mathcal{P}$ a été affinée par $S = f^{-1}(\mathcal{X})$, alors $\mathcal{P}$ est stable. La classe $\mathcal{X}$ peut donc être supprimée de *Pivots*. Soit $\mathcal{Y}$ une classe affinée

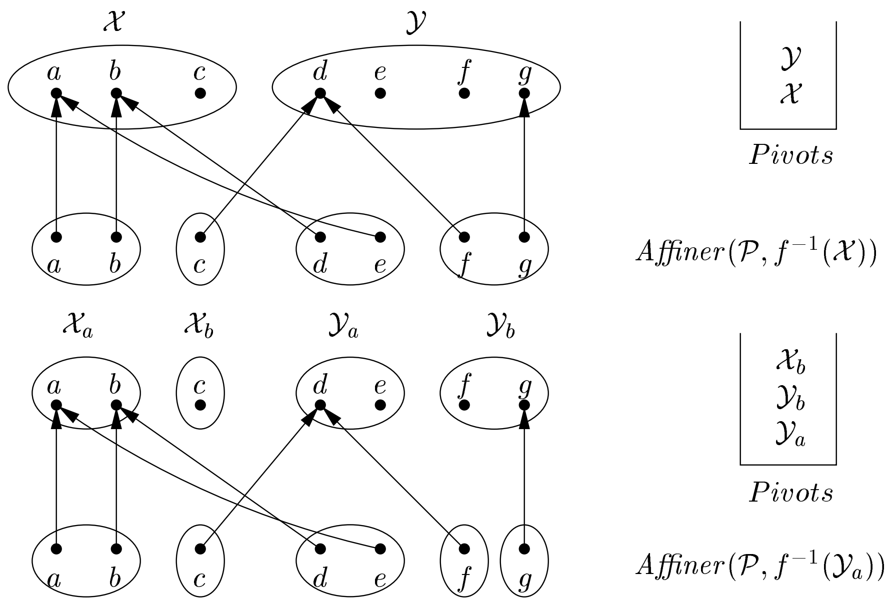


FIG. 7.2 – *Après les deux étapes Affiner*$(\mathcal{P}, f^{-1}(\mathcal{X}))$ *et Affiner*$(\mathcal{P}, f^{-1}(\mathcal{Y}_a))$*, nous obtenons la partition fonctionnelle compatible maximum.*

par $S = f^{-1}(\mathcal{X})$. Il y a deux situations possibles pour $\mathcal{Y}$:

- $\mathcal{Y} \notin$ *Pivots* : Alors $\mathcal{P}$ est stable pour $f^{-1}(\mathcal{Y})$. Donc $\mathcal{Y}$ n'a pas été coupée depuis que $\mathcal{P}$ a été affinée par $f^{-1}(\mathcal{Y})$. Le lemme 7.4 permet d'appliquer la règle de *"la plus petite moitié"* : *i.e.* n'ajouter à *Pivots* que la plus petite classe parmi $\mathcal{Y}_a$ et $\mathcal{Y}_b$.

- $\mathcal{Y} \in$ *Pivots* : Rappelons que *Pivots* contient un pointeur sur $\mathcal{Y}$ (voir chapitre 5). Ce pointeur adresse donc désormais la classe $\mathcal{Y}_b$ (les éléments de $\mathcal{Y}_a$ ont été supprimés de $\mathcal{Y}$). Puisque $\mathcal{P}$ n'était pas nécessairement stable pour $f^{-1}(\mathcal{Y})$, nous devons ajouter $\mathcal{Y}_a$ à *Pivots*. Notons que $f^{-1}(\mathcal{Y}) = f^{-1}(\mathcal{Y}_a) \cup f^{-1}(\mathcal{Y}_b)$ (union disjointe). Donc globalement, aucun travail supplémentaire n'est rajouté pour les futures étapes.

Présentons maintenant l'algorithme. Notons que la procédure *ClassePivot* utilisée est la procédure 11. Elle ne calcule qu'un ensemble pivot par élément de *Pivots*.

Algorithme 16: Partition fonctionnelle maximum

Données : une partition $\mathcal{Q}$ sur E et une fonction $f : E \to E$

Résultat : $\mathcal{P}$, la partition fonctionnelle maximum pour la partition $\mathcal{Q}$ et la fonction f

début

  pour chaque *élément* $x \in E$ **faire** calculer l'ensemble $f^{-1}(x)$;

  exécuter l'algorithme 9 avec la procédure 11 et les définitions suivantes (les éléments de *Pivots* sont des classes de $\mathcal{P}$);

fin

Procedure *InitPartition* ($\mathcal{P}$)

  au premier appel :

    $\mathcal{P} \leftarrow \mathcal{Q}$;

    ajouter toutes les classes de $\mathcal{P}$ à *Pivots*;

  au second appel :

    fin du processus, **retourner** $\mathcal{P}$;

Procedure *EnsemblePivot* ($p = (x, \mathcal{X})$)

  retourner $f^{-1}(x)$;

Procedure *InsertàDroite* ($\mathcal{X}, \mathcal{X}_a, p$)

  retourner vrai;

Procedure *AjoutPivot* ($\mathcal{X}, \mathcal{X}_a$)

  soit $\mathcal{Z}$ la plus petite classe entre $\mathcal{X}$ et $\mathcal{X}_a$;

  si $\mathcal{X} \in$ *Pivots (i.e.* $\mathcal{X}$ *a été ajouté à Pivots avant que* $\mathcal{X}_a$ *soit supprimée de* $\mathcal{X}$*)* **alors** ajouter $\mathcal{X}_a$ à *Pivots*;

  sinon ajouter $\mathcal{Z}$ à *Pivots*;

Invariant : *La partition* $\mathcal{P}$ *vérifie la propriété* A*; si* $\mathcal{P}$ *ne vérifie pas* B*, alors il existe un pivot dans Pivots qui affine strictement* $\mathcal{P}$*.*

- ***A:*** *Toute partition fonctionnelle pour la partition* $\mathcal{Q}$ *et la fonction* f *est plus fine que* $\mathcal{P}$*;* $\mathcal{P}$ *est compatible avec* $\mathcal{Q}$*.*
- ***B:*** $\mathcal{P}$ *est stable pour* $f^{-1}(\mathcal{X})$ *pour chaque classe* $\mathcal{X}$ *de* $\mathcal{P}$*.*

Preuve : Faisons une induction pour montrer que chaque étape d'affinage préserve la propriété A. Cette propriété est vraie après le premier appel à la procédure *InitPartition* car $\mathcal{P} = \mathcal{Q}$ donc $\mathcal{P}$ compatible avec $\mathcal{Q}$ et toute partition fonctionnelle pour $\mathcal{Q}$ est plus fine que $\mathcal{P}$. Supposons que la propriété soit vraie avant chaque étape d'affinage.

Soient $\mathcal{P}$ la partition courante et $\mathcal{Y}$ une classe de $\mathcal{P}$ telle que $\mathcal{P}$ ne soit pas stable pour $f^{-1}(\mathcal{Y})$. Nous notons alors $\mathcal{P}' = \mathit{Affiner}(\mathcal{P}, f^{-1}(\mathcal{Y}))$. Tout d'abord, la structure de données utilisée garantit que $\mathcal{P}'$ soit compatible avec $\mathcal{P}$. Donc par induction, $\mathcal{P}'$ est compatible avec $\mathcal{Q}$.

Par hypothèse d'induction, toute partition fonctionnelle $\mathcal{R}$ est plus fine que $\mathcal{P}$. Montrons que $\mathcal{R}$ est plus fine que $\mathcal{P}'$. Supposons qu'il existe une classe $\mathcal{X}^r \in \mathcal{R}$ qui ne soit incluse dans aucune classe de $\mathcal{P}'$. Puisque $\mathcal{R}$ est plus fine que $\mathcal{P}$, la classe $\mathcal{X}^p \in \mathcal{P}$ qui contenait $\mathcal{X}^r$ a été coupée en deux sous-classes $\mathcal{X}^p_a = \mathcal{X}^p \cap f^{-1}(\mathcal{Y})$ et $\mathcal{X}^p_b = \mathcal{X}^p \setminus f^{-1}(\mathcal{Y})$. Puisque X^r n'est inclus ni dans $\mathcal{X}^p_a$ ni dans $\mathcal{X}^p_b$, $\mathcal{X}^r$ contient des éléments de ces deux classes. Autrement dit, $\mathcal{X}^r$ n'est pas stable pour $f^1(\mathcal{Y})$, donc $\mathcal{R}$ n'est pas une partition fonctionnelle : contradiction.

Montrons que le processus termine avec le résultat souhaité : *i.e.* lorsque $Pivots = \emptyset$ alors la propriété B est vérifiée. Si une classe $\mathcal{Y}$ de la partition $\mathcal{P}$ n'appartient pas à $Pivots$, cela peut correspondre à deux situations différentes :

1. Soit $\mathcal{P}$ a été affinée par $f^{-1}(\mathcal{Y})$ et depuis cette étape, $\mathcal{Y}$ n'a jamais été affinée. Donc $\mathcal{P}$ est stable pour $f^{-1}(\mathcal{Y})$.

2. Soit $\mathcal{Y}$ a été obtenue en affinant une classe $\mathcal{X}$ en $\mathcal{Y}$ et $\mathcal{Z}$, où $|\mathcal{Z}| \leqslant |\mathcal{Y}|$ et $\mathcal{Y}$ n'a jamais été affinée depuis. C'est alors la classe $\mathcal{Z}$ qui a été ajoutée à $Pivots$. Le lemme 7.4 montre qu'après affinage de $\mathcal{P}$ par $\mathcal{Z}$, $\mathcal{P}$ est stable pour $\mathcal{Y}$.

Le processus s'arrête lorsque $Pivots$ est vide, alors $\mathcal{P}$ est stable pour $f^{-1}(\mathcal{X})$ quel que soit $\mathcal{X} \in \mathcal{P}$: $\mathcal{P}$ est donc une partition fonctionnelle pour $\mathcal{Q}$ et la fonction f.

De plus, la propriété A montre que $\mathcal{P}$ est plus grossière que toutes les partitions fonctionnelles pour $\mathcal{Q}$. Puisque c'est elle-même une partition fonctionnelle, c'est l'unique partition fonctionnelle maximum (*i.e.* avec la moins de classes). Le lemme 7.3 est donc prouvé. Puisque A montre aussi que $\mathcal{P}$ est compatible avec $\mathcal{Q}$, alors $\mathcal{P}$ est une partition fonctionnelle maximum pour la fonction f compatible avec $\mathcal{Q}$. □

Analyse de complexité : Le pré-calcul des ensembles $f^{-1}(x)$ pour $x \in E$ peut se faire en $O(n)$ où $n = |E|$. Le coût de la procédure *InitPartition* est $O(n)$, elle n'est appelée que deux fois. Celui de la procédure *InsertàDroite* est constant. Il suffit donc d'évaluer la complexité des procédures *EnsemblePivot* et *AjoutPivot* ainsi que le nombre de fois où elles sont utilisées.

Le calcul de l'ensemble pivot S (procédure *AjoutPivot*) se fait en la taille de S. Nous savons déjà que $\mathit{Affiner}(\mathcal{P}, S)$ demande $O(|S|)$. Il reste à analyser le nombre de fois où l'ensemble $f^{-1}(x)$ peut être utilisé pour affiner la partition courante.

Intéressons-nous donc à la procédure *AjoutPivot*. Soit $\mathcal{X}$ une classe de la partition $\mathcal{P}$, affinée à une étape donnée en $\mathcal{X}_a = \mathcal{X} \cap S$ et $\mathcal{X}_b = \mathcal{X} \setminus X_a = \mathcal{X} \setminus S$ où S est l'ensemble pivot. Rappelons que la structure de données en liste doublement chaînée et pointeurs de la partition permet de calculer $\mathcal{X}_a$ et $\mathcal{X}_b$ en $O(|\mathcal{X}_a|)$ et le pointeur de $\mathcal{X}_b$ est celui de l'ancienne classe $\mathcal{X}$. Donc si la classe $\mathcal{X}$ était déjà présente dans *Pivots*, le pointeur correspondant n'est pas retiré de *Pivots* et correspond désormais à $\mathcal{X}_b$. Ainsi l'ajout de $\mathcal{X}_a$ à *Pivots* n'augmente pas la taille globale des ensembles pivots qui sont utilisés ($f^{-1}(\mathcal{X}) = f^{-1}(\mathcal{X}_b) \cup f^{-1}(\mathcal{X}_a)$).

Le cas contraire est plus intéressant : la classe $\mathcal{X}$ n'appartient pas à *Pivots*. Dans ce cas, nous appliquons la règle de Hopcroft : seule la classe correspondant au plus petit ensemble parmi $f^{-1}(\mathcal{X}_a)$ et $f^{-1}(\mathcal{X}_b)$ est ajouté à *Pivots*. Puisque $f^{-1}(\mathcal{X})$ est l'union des $f^{-1}(x)$ pour $x \in \mathcal{X}$, ce choix garantit que pour chaque élément $x \in E$, l'ensemble $f^{-1}(x)$ est visité au plus $\log n$ où n est le nombre de classes. Comme la taille globale des ensembles $f^{-1}(x)$ est n, la complexité de l'algorithme 16 est $O(n \log n)$. □

7.2 Minimisation d'automates déterministes

La problème de la partition fonctionnelle maximum a été présentée dans [AHU74] comme un cas particulier de la minimisation d'automates finis déterministes. Nous indiquons comment l'algorithme 16 se généralise. Une autre généralisation de la partition fonctionnelle maximum sera présentée dans la prochainle section : le calcul de la partition maximum respectant non plus une fonction mais une relation binaire (*i.e.* un élément peut avoir plusieurs images).

Définition 7.6 *Nous appelons* ***automate fini déterministe*** *un quintuplet* $\mathcal{A} = (Q, i, T, \Sigma, \delta)$ *où :*

- Q *est un ensemble fini d'états*
- i *est l'état initial*
- $T \subseteq Q$ *est l'ensemble des états terminaux*
- Σ *est un alphabet fini*
- $\delta : Q \times \Sigma \to Q$ *est la fonction de transition*

Notation 7.7 *Soient* $a \in \Sigma$ *et* $q \in Q$*, alors la transition* $\delta(q, a) = q'$ *sera notée* $q.a = q'$*. Soit* Σ^* *l'ensemble des mots sur l'aphabet* Σ*. Alors si* $w = [a_1 a_2 \ldots a_k] \in \Sigma^*$*, l'état* $q' = q.w$ *est l'état atteint par les transitions* $\delta(\delta(\ldots\delta(q, a_1)\ldots), a_k)$*.*

Nous ferons deux hypothèses supplémentaires sur les automates. Elle ne sont pas indispensables, mais elles simplifient l'algorithme. Les automates sont donc :

– *complets* : $\forall q \in Q$ et $\forall a \in \Sigma$, il existe une transition $q.a$.

– *accessibles* : tout état terminal peut être atteint depuis l'état initial.

Définition 7.8 *Soit $\mathcal{A} = (Q, i, T, \Sigma, \delta)$ un automate. Pour chaque état $q \in Q$, $L_q(\mathcal{A}) = \{w \in \Sigma^* : q.w \in T\}$ est le langage reconnu par $\mathcal{A}$ depuis l'état q.*

Minimiser un automate consiste à diminuer le nombre d'états de telle sorte qu'il n'existe pas deux états reconnaissant le même langage.

Définition 7.9 *[AHU74, BBC92] Deux états p et q sont* Nérode équivalents *si et seulement si $L_p(\mathcal{A}) = L_q(\mathcal{A})$. Nous le noterons $p \sim q$.*

Nous dirons qu'un mot $w \in \Sigma^*$ *sépare* deux états p et q si $p.w \in T$ et $q.w \notin T$. Deux états sont donc *Nérode équivalents* si et seulement si aucun mot ne les sépare. La relation de Nérode est une relation d'équivalence. Nous allons donc chercher une partition des états de Q en classes d'équivalence de Nérode. Comme dans le cas du calcul des sommets jumeaux, la partition que nous cherchons n'est pas une partition ordonnée.

Nous rappelons maintenant très brièvement comment les classes d'équivalence de Nérode peuvent être obtenues. Pour plus de détails, le lecteur intéressé peut se référer à [BBC92]. Pour chaque état p, considérons le langage des mots de longueur inférieure à k reconnu depuis p. Nous le notons :

$$L_p^k(\mathcal{A}) \quad = \quad \{w \in L_p(\mathcal{A}) : |w| \leqslant k\}$$

Considérons la relation d'équivalence : $$p \sim_k q \Leftrightarrow L_p^k(\mathcal{A}) = L_q^k(\mathcal{A})$$
D'une part, la relation d'équivalence de Nérode est l'intersection des relations d'équivalence $\sim_k$ pour tout k. Mais surtout la relation d'équivalence $\sim_k$ peut s'exprimer en fonction de $\sim_{k-1}$: $$p \sim_k q \Leftrightarrow p \sim_{k-1} q \text{ et } \forall a \in \Sigma,\ p.a \sim_{k-1} q.a$$

Cette correspondance est très intéressante car elle signifie qu'il n'est pas nécessaire de calculer $L_p^k(\mathcal{A})$ et $L_q^k(\mathcal{A})$ pour se rendre compte que $p \sim q$. Il suffit de séparer itérativement les états à l'aide des langages $L_p^1(\mathcal{A})$: si à partir de deux états p et q, la lettre a mène dans deux états différents p' et q' qui ne sont pas *Nérode équivalents*, alors p et q ne sont pas Nérode équivalents.

Notation 7.10 *Soient un ensemble E d'états d'un automate $\mathcal{A} = (Q, i, T, \Sigma, \delta)$ et a une lettre de l'alphabet Σ. L'ensemble $a^{-1}E$ désigne l'ensemble des états $q \in Q$ tels que $q.a \in E$.*

Clairement les états terminaux T et non-terminaux $Q \setminus T$ ne sont pas Nérode équivalents. La partition initiale sera donc composée de T et $Q \setminus T$. Chaque étape d'affinage consiste à affiner les classes de la partition par un ensemble pivot $a^{-1}\mathcal{X}$ où $a \in \Sigma$ et $\mathcal{X} \in \mathcal{P}$. Si $a^{-1}\mathcal{X}$ affine strictement la partition, cela signifie qu'il existe un mot commençant par a et séparant deux états de la même classe de $\mathcal{P}$. Pour minimiser l'espace mémoire, nous ne garderons comme information dans *Pivots* que des couples *(état,lettre)*. L'ensemble pivot sera calculé au dernier moment.

Avant de présenter une implémentation de l'algorithme de Hopcroft basée sur notre formalisme (algorithme 18) rappelons un dernier résultat nécessaire à sa compréhension. La preuve de cette propriété peut être consultée dans [BBC92].

Lemme 7.11 *Soit $\mathcal{P}$ une partition des états de l'automate $\mathcal{A}$ telle que tout couple d'états Nérode équivalents appartient à une même classe. Soient $\mathcal{X}$ et $\mathcal{Y}$, deux classes distinctes de $\mathcal{P}$ et a, b deux lettres de σ.*

1. *Affiner $\mathcal{P}$ par $a^{-1}\mathcal{X}$ puis $b^{-1}\mathcal{Y}$ est équivalent à affiner $\mathcal{P}$ par $b^{-1}\mathcal{Y}$ puis $a^{-1}\mathcal{X}$.*
2. *Si $\mathcal{X}_1, \mathcal{X}_2$ est une partition de $\mathcal{X}$, alors affiner $\mathcal{P}$ par $a^{-1}\mathcal{X}_1$ puis $a^{-1}\mathcal{X}_2$ est équivalent à affiner $\mathcal{P}$ par $a^{-1}\mathcal{X}$ puis $a^{-1}\mathcal{X}_1$ (ou $a^{-1}\mathcal{X}_2$).*

La gestion des ensembles pivots est simple. L'ordre sur le choix des pivots n'a pas d'importance et le point 2 du lemme précédent est à rapprocher du lemme 7.4. L'automate étant complet, chaque lettre $a \in \Sigma$ correspond à une fonction de Q dans Q. La minimisation d'automates est donc le calcul de la partition maximum compatible avec $\mathcal{P} = \{Q \setminus T, T\}$ pour un ensemble de fonctions : c'est donc une généralisation de la partition fonctionnelle maximum. Pour tenir compte de cette différence, nous devons réécrire la procédure *ClassePivot*.

Procédure 17: *ClassePivot* $(\mathcal{E} = (\mathcal{X}, a))$

Données : un élément $\mathcal{E} = (\mathcal{X}, a)$ de *Pivots*
Résultat : Une partition stable pour l'ensemble pivot généré par $\mathcal{E}$
début
    $S \leftarrow \emptyset$;
    pour chaque $x \in \mathcal{X}$ **faire**
        $p \leftarrow (x, a)$;
        $S \leftarrow S \cup$ *EnsemblePivot* (p);
    affiner$(\mathcal{P}, S)$;
fin

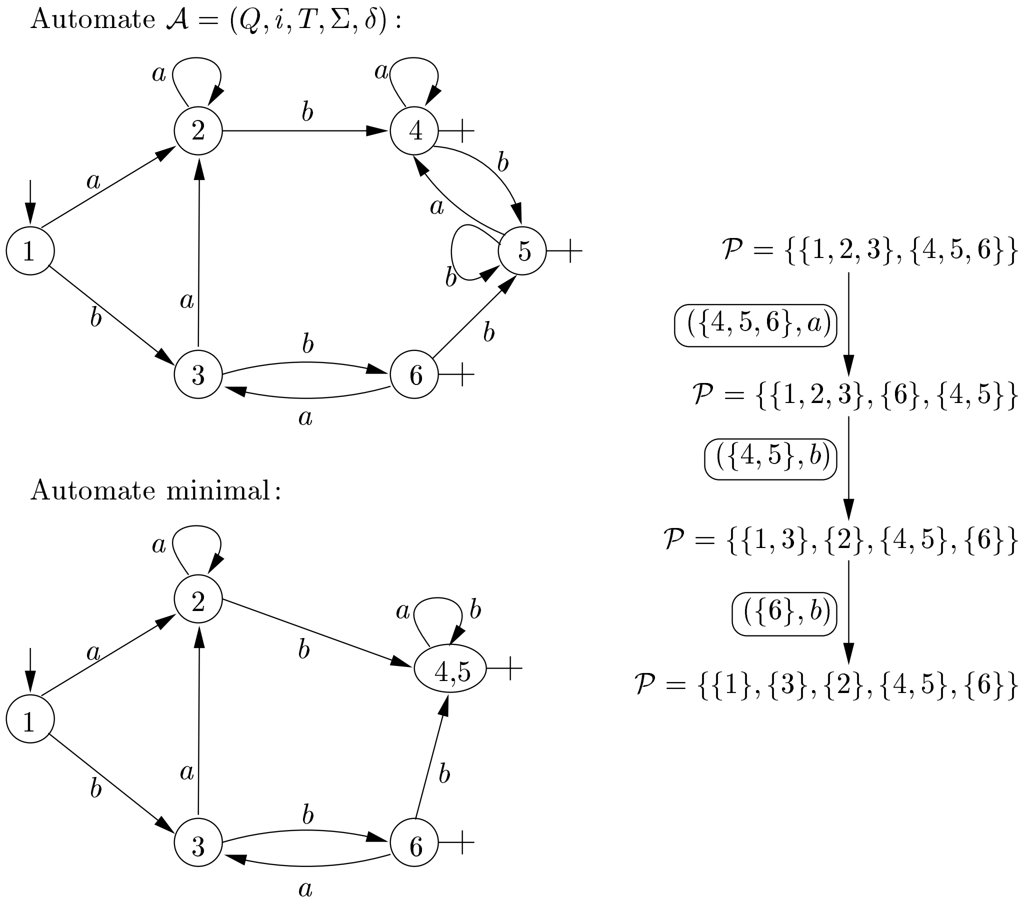


FIG. 7.3 – *Réduction d'un automate fini déterministe. Les états* 4 *et* 5 *sont Nérodes équivalents.*

La gestion de *Pivots* est alors la même que dans le calcul de la partition fonctionnelle maximum. Il faut juste tenir compte de toutes les lettres de l'alphabet Σ. La figure 7.3 décrit une exécution partielle de l'algorithme :

- Initialement $Pivots = \{(\{4,5,6\},a),(\{4,5,6\},b)\}$.
- Le premier affinage se fait selon le pivot $(\{4,5,6\},a)$. L'ensemble pivot associé est $\{4,5\}$. La classe $\{4,5,6\}$ est donc coupée en $\{4,5\}$ et $\{6\}$. Les couples $(\{6\},b)$ et $(\{6\},a)$ sont rajoutés à *Pivots*.
- Le second affinage se fait selon $(\{4,5\},b)$, l'ensemble associé est $\{2,4,5,6\}$. La classe $\{1,2,3\}$ est coupée en $\{1,3\}$ et $\{2\}$. Les couples $(\{2\},a)$ et $(\{2\},b)$ sont ajoutés à *Pivots*.

– Le troisième affinage se fait selon $(\{6\}, b)$, l'ensemble associé est $\{3\}$. La classe $\{1, 3\}$ est coupée en $\{1\}$ et $\{3\}$. Les couples $(\{1\}, a)$ et $(\{1\}, b)$ sont ajoutés à $Pivots$.

Après le troisième affinage, $Pivots$ n'est pas vide, mais la partition est stable. Nous ne décrivons pas le reste. L'algorithme 18 détaille le processus.

Notation 7.12 *Pour éviter toute confusion avec les lettres de Σ, lorsqu'une classe $\mathcal{X}$ est affinée par S, nous noterons $X_1 = \mathcal{X} \cap S$ et $\mathcal{X}_2 = \mathcal{X} \setminus S$.*

Algorithme 18: Minimisation d'automates finis déterministes

Données : Un automate $\mathcal{A} = (Q, i, T)$ complet, accessible, fini et déterministe

Résultat : Une partition $L(\mathcal{P})$ des états en classes d'équivalence de Nérode

début

  pour chaque *couple (état,lettre)* (q, a) **faire** calculer l'ensemble des états $a^{-1}q$;

  exécuter l'algorithme 9 avec la procédure 17 les définitions suivantes (les éléments de $Pivots$ sont les couples (classe,lettre));

fin

Procedure *InitPartition* $(L(\mathcal{P}))$

  au premier appel :

    $\mathcal{P} \leftarrow (Q \setminus T, T)$;

    AjoutPivot $(Q \setminus T, T)$;

  au second appel :

    fin du processus, **retourner** $\mathcal{P}$;

Procedure *EnsemblePivot* $(p = (q, a))$

  retourner $a^{-1}q$;

Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_1, p)$

  retourner vrai;

Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_1)$

  soit $\mathcal{Z}$ la plus petite classe entre $\mathcal{X}$ et $\mathcal{X}_1$;

  pour chaque $a \in \Sigma$ **faire**

    si $(\mathcal{X}, a) \in Pivots$ *(i.e.* $(\mathcal{X}, a)$ *a été ajouté à* $Pivots$ *avant que* $\mathcal{X}_1$ *soit supprimée de* $\mathcal{X}$*)* **alors** ajouter $(\mathcal{X}_1, a)$ à $Pivots$;

    sinon ajouter $(\mathcal{Z}, a)$ à $Pivots$;

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Si deux états p et q sont Nérode équivalents, alors ils appartiennent à la même classe.*
- ***B:*** *$\mathcal{P}$ est stable pour $a^{-1}\mathcal{X}$ pour chaque classe $\mathcal{X} \in \mathcal{P}$ et chaque lettre $a \in \Sigma$.*

Preuve : Le mot vide sera noté ε. Nous montrons par induction que la propriété A est préservée par une étape d'affinage. Tout d'abord les états terminaux de T et les états non-terminaux de $Q \setminus T$ ne sont clairement pas *Nérode équivalents* puisque ε les sépare. Donc A est vraie après la procédure *InitPartition*. Par induction, supposons qu'elle le soit avant l'affinage de $\mathcal{P}$ par un ensemble pivot $a^{-1}\mathcal{X}$, où $a \in \Sigma$ et où $\mathcal{X} \in \mathcal{P}$, et montrons qu'elle l'est toujours après.

Supposons que deux états *Nérode équivalents* p et q soient séparés au cours de cette étape d'affinage. Cela signifie que $p' = p.a$ et $q' = q.a$ appartiennent à deux classes différentes. Par hypothèse d'induction p' et q' ne sont pas *Nérode équivalents.* Il existe donc un mot $w \in L_{p'}(\mathcal{A})$ et $w \notin L_{q'}(\mathcal{A})$. Autrement dit, $a.w \in L_p(\mathcal{A})$ et $a.w \notin L_q(\mathcal{A})$. Donc p et q ne sont pas *Nérode équivalents* : contradiction.

Montrons maintenant que si la propriété B est fausse, alors il existe dans $Pivots$ des éléments qui affinent strictement $\mathcal{P}$. Supposons que $\mathcal{P}$ ne soit pas stable pour l'ensemble $a^{-1}\mathcal{X}$ où $a \in \Sigma$ et $\mathcal{X}$ est une classe de $\mathcal{P}$. Il existe donc deux états p et q appartenant à une même classe tels que $p.a \in \mathcal{X}$ et $q.a \notin \mathcal{X}$. Considérons la première étape où $p.a$ et $q.a$ ont été séparés dans deux classes différentes. Alors la plus petite de ces deux classes, notée $\mathcal{Z}$, a été ajoutée à $Pivots$ avec la lettre a. La classe $\mathcal{Z}$ contient soit $p.a$ soit $q.a$ (mais pas les deux). Donc au couple $(\mathcal{Z}, a)$ correspond un ensemble pivot $S = a^{-1}\mathcal{Z}$ tel que $p \in S$ et $q \notin S$ (ou l'inverse). C'est pourquoi S affine strictement la partition $\mathcal{P}$. Notons que lorsqu'une classe $\mathcal{X}$ est coupée en $\mathcal{X}_1$ et $\mathcal{X}_2$, si $(\mathcal{X}, a)$ appartenait à $Pivots$ avant l'affinage, alors $(\mathcal{X}_1, a)$ et $(\mathcal{X}_2, a)$ appartiennent à $Pivots$ après. Donc $Pivots$ n'est pas vide.

Le processus d'affinage s'arête lorsque $Pivots = \emptyset$. Il nous reste à montrer que la partition finale correspond aux classes d'équivalence de *Nérode*. Supposons que les états p et q appartiennent à la même classe de la partition finale $\mathcal{P}$ et qu'il existe un mot $w \in \Sigma^*$ tel que $w \notin L_q(\mathcal{A})$ et $w \in L_p(\mathcal{A})$ (ou l'inverse). Soit u le plus long préfixe de w tel que $p.u$ et $q.u$ soient dans la même classe. Notons a la lettre immédiatement après u dans w. Soit $\mathcal{X}$ la classe contenant $q.ua$. Alors la partition n'est pas stable pour l'ensemble $a^{-1}\mathcal{X}$, la propriété B est donc fausse. Nous avons vu que dans ce cas $Pivots$ n'est pas vide. Donc $\mathcal{P}$ ne peut pas être la partition finale : contradiction. $\square$

Analyse de complexité : Cette preuve est similaire à celle faite pour la complexité de l'algorithme 16. Le pré-calcul des ensembles $a^{-1}q$ est linéaire. Le coût de la procédure *InitPartition* est linéaire et elle n'est appelée que deux fois. La procédure *InsertàDroite* a un coût constant, il suffit donc d'évaluer la complexité des procédures *EnsemblePivot* et *AjoutPivot* ainsi que le nombre de fois où elles sont utilisées.

La procédure *EnsemblePivot* calcule l'ensemble pivot $a^{-1}\mathcal{X}$ en $O(|a^{-1}\mathcal{X}|)$. Cette procédure est appelée à chaque étape d'affinage de la partition. Puisque l'affinage se fait aussi en $O(|a^{-1}\mathcal{X}|)$, il faut regarder le nombre de fois qu'un ensemble pivot peut être utilisé. Autrement dit le nombre de fois qu'un couple $(\mathcal{X}, a)$ peut être ajouté dans *Pivots*.

Intéressons-nous donc à la procédure *AjoutPivot*. Soit $\mathcal{X}$ une classe de la partition $\mathcal{P}$, affinée à une étape donnée en $\mathcal{X}_1$ et $\mathcal{X}_2 = \mathcal{X} \setminus \mathcal{X}_1$. Rappelons que la structure de données de la partition permet de calculer $\mathcal{X}_1$ et $\mathcal{X}_2$ en $O(|\mathcal{X}_1|)$.

Donc lorsque la classe $\mathcal{X}$ est affinée, si le couple $(\mathcal{X}, a)$ pour $a \in \Sigma$ appartient à *Pivots*, alors il correspond à $(\mathcal{X}_2, a)$ ($\mathcal{X}$ a été modifié en $\mathcal{X} \setminus \mathcal{X}_1 = \mathcal{X}_2$). Ainsi l'ajout de $(\mathcal{X}_1, a)$ n'augmente pas la taille globale des ensembles pivots qui sont utilisés.

Le cas contraire est plus intéressant (le couple $(\mathcal{X}, a)$ n'appartient pas à *Pivots*). Grâce au lemme 7.11, nous pouvons appliquer la règle de Hopcroft : seul le couple correspondant au plus petit ensemble parmi $a^{-1}\mathcal{X}_2$ et $a^{-1}\mathcal{X}_1$ est ajouté à *Pivots*. Puisque $a^{-1}\mathcal{X}$, où $a \in \Sigma$ est l'union des $a^{-1}q$ pour $q \in \mathcal{X}$, ce choix garantit que pour chaque lettre a et chaque état q, l'ensemble $a^{-1}q$ est traversé au plus $\log n$ où n est le nombre d'états. Comme il y a k lettres, la complexité de l'algorithme 18 est $O(nk \log n)$.

L'espace mémoire nécessaire est borné par la taille de *Pivots*. Or *Pivots* contient au plus $n'k$ couples où n' est le nombre d'éléments de la partition courante. Or n' est borné par n, la taille de *Pivots* est donc $O(nk)$. □

7.3 Partition relationnelle maximum

Dans cette section, nous étudions une autre généralisation de la partition fonctionnelle maximum : la partition relationnelle maximum. Ce problème a été présenté par Kanellakis et Smolka [KS83] dans le cadre des systèmes de calculs communiquants. Il s'agit cette fois de trouver la partition maximum respectant non plus une fonction mais une relation binaire R sur un ensemble E.

La principale différence avec la partition fonctionnelle maximum est qu'un élément $x \in E$ peut avoir plusieurs images par R. Dans le cas des fonctions, nous avions $|f(x)| = 1$. Nous verrons dans un premier temps toutes les conséquences de cette différence. Puis nous montrerons comment modifier légèrement l'algorithme 9 pour implémenter l'algorithme proposé par Paigue et Tarjan [PT87].

Définition 7.13 *Soient une partition $\mathcal{Q} = \{\mathcal{B}_1, \ldots, \mathcal{B}_p\}$ d'un ensemble E et une relation binaire $R : E \to 2^E$. La **partition relationnelle** pour la partition $\mathcal{Q}$ et pour la relation R est une partition $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_q\}$ de E telle que :*

1. *$\mathcal{P}$ est* plus fine *que $\mathcal{Q}$: i.e. $\mathcal{P} \preccurlyeq \mathcal{Q}$.*
2. *$\mathcal{P}$ est* stable *pour R : i.e. si x et y appartiennent à $\mathcal{X}_i \in \mathcal{P}$ et si $R(x) \cap \mathcal{X}_j \neq \emptyset$ où $\mathcal{X}_j \in \mathcal{P}$ alors $R(y) \cap \mathcal{X}_j \neq \emptyset$.*

La remarque 7.2 est toujours valable : nous calculons des partitions compatibles avec la partition initiale $\mathcal{Q}$. De plus, il y a aussi unicité de la partition relationnelle ayant le moins de classes :

Lemme 7.14 *Il existe une unique partition relationnelle maximum de $\mathcal{Q}$.*

Remarque 7.15 *Puisque $|R(x)| > 1$ est possible, le lemme 7.4 n'est plus applicable*

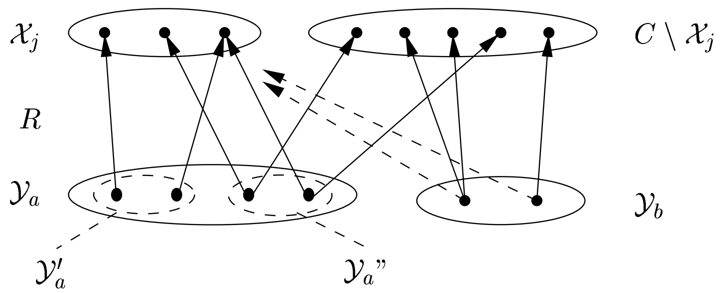


FIG. *7.4* – La classe $\mathcal{Y}_a$ est strictement affinée par $R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ en $\mathcal{Y}'_a = \mathcal{Y}_a \cap R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ et $\mathcal{Y}_a" = \mathcal{Y}_a \setminus R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ (les flèches en pointillés indiquent qu'un élément n'a pas d'image par f dans un ensemble).

Soit $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$ une union de classes de $\mathcal{P}$ telle que $\mathcal{P}$ n'est pas stable pour $R^{-1}(\mathcal{C})$. Alors $\mathcal{P}' = Affiner(\mathcal{P}, R^{-1}(\mathcal{X}_j))$, $i \leqslant j \leqslant k$ n'est pas nécessairement stable pour $R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$. Supposons que la classe $\mathcal{Y}$ de $\mathcal{P}$ soit affinée par l'ensemble pivot $R^{-1}(\mathcal{X}_j)$ en $\mathcal{Y}_a = \mathcal{Y} \cap R^{-1}(\mathcal{X}_j)$ et $\mathcal{Y}_b = \mathcal{Y} \setminus R^{-1}(\mathcal{X}_j)$:

- *Puisque $\mathcal{Y}$ est stable pour $\mathcal{C}$, les éléments de $\mathcal{Y}$ ont au moins une image dans C. Or les éléments de $\mathcal{Y}_b$ n'ont pas d'image par R dans $\mathcal{X}_j$, ils doivent donc en avoir une dans $\mathcal{C} \setminus \mathcal{X}_j$. Donc $\mathcal{Y}_b$ est stable pour $R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$.*

– *En revanche, il est possible que certains éléments de $\mathcal{Y}_a$ aient aussi une image dans $\mathcal{C} \setminus \mathcal{X}_j$ ($|R(x)| \geqslant 1$). Donc $\mathcal{Y}_a$ n'est pas nécessairement stable par $R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ (voir figure 7.4) et est affinée en $\mathcal{Y}'_a = \mathcal{Y}_a \cap R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ et $\mathcal{Y}_a" = \mathcal{Y}_a \setminus R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$.*

Les conditions d'application de la règle de Hopcroft décrite à la section 1 ne sont donc plus vérifiées. Cependant Paigue et Tarjan [PT87] ont mis au point une structure de données qui permet d'affiner une partition en fonction de $R^{-1}(\mathcal{X}_j)$ et $R^{-1}(\mathcal{C} \setminus \mathcal{X}_j)$ en n'utilisant que l'ensemble pivot $S = R^{-1}(\mathcal{X}_j)$. Mais cela implique que l'étape d'affinage selon $\mathcal{C} \setminus \mathcal{X}_j$ doit suivre celle selon $\mathcal{X}_j$. Nous devons donc légèrement modifier la procédure *Affiner* (voir procédure 19).

Procédure 19: *Affiner*$(\mathcal{P}, S)$

Données : une partition $\mathcal{P}$, un ensemble pivot S et une relation binaire E
Résultat : une partition $\mathcal{P}'$ compatible avec $\mathcal{P}$ et stable pour S
début
  soit M l'ensemble des classes intersectant strictement S (*i.e.* intersectant S et non incluses dans S);
  pour chaque *classe* $\mathcal{X} \in M$ **faire**
    affinage par $S = R^{-1}(\mathcal{Z})$
      soit $\mathcal{X}_a$ les éléments de $\mathcal{X}$ appartenant à S;
      supprimer $\mathcal{X}_a$ de $\mathcal{X}$;
      si *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$ **alors**
        insérer $\mathcal{X}_a$ à droite de $\mathcal{X}$;
      sinon insérer $\mathcal{X}_a$ à gauche de $\mathcal{X}$;
      AjoutPivot $(\mathcal{X}, \mathcal{X}_a)$;
    affinage par $R^{-1}(\mathcal{C} \setminus Z)$ *où* $\mathcal{C}$ *est la classe de* $\mathcal{P}'$ *contenant* $\mathcal{Z}$
      soit $\mathcal{X}'_a$ l'ensemble des éléments $x \in \mathcal{X}_a$ tels que $R(x) \cap \mathcal{Z} = R(x) \cap \mathcal{C}$;
      supprimer $\mathcal{X}'_a$ de $\mathcal{X}_a$;
      si *InsertàDroite* $(\mathcal{X}_a, \mathcal{X}'_a, p)$ **alors**
        insérer $\mathcal{X}'_a$ à droite de $\mathcal{X}_a$;
      sinon insérer $\mathcal{X}'_a$ à gauche de $\mathcal{X}_a$;
fin

Présentons les grandes lignes des idées de Paigue et Tarjan. Soit $\mathcal{Y}$ une classe de $\mathcal{P}$ affinée par $R^{-1}(\mathcal{X}_j)$ en $\mathcal{Y}_a = \mathcal{Y} \cap R^{-1}(\mathcal{X}_j)$ et $\mathcal{Y}_b = \mathcal{Y} \setminus R^{-1}(\mathcal{X}_j)$. Nous savons que seule la sous-classe $\mathcal{Y}_a = \mathcal{Y} \cap R^{-1}(\mathcal{X}_j)$ peut être affinée par $R^{-1}(\mathcal{C} \setminus \mathcal{X}_i)$. En utilisant une procédure d'affinage classique, nous chercherions les éléments de $\mathcal{Y}_a$ ayant une image par R dans $\mathcal{C} \setminus \mathcal{X}_j$. Le problème consiste donc à identifier ces éléments : *i.e.* les éléments $y \in \mathcal{Y}_a$ tels que $R(y) \cap \mathcal{C} \neq R(y) \cap \mathcal{X}_j$. Supprimer de tels éléments de $\mathcal{Y}_a$ pour scinder $\mathcal{Y}_a$ en deux est strictement équivalent à supprimer les éléments $y' \in \mathcal{Y}_a$ tels que $R(y') \cap \mathcal{C} = R(y') \cap \mathcal{X}_j$. Paigue et Tarjan ont utilisé cette seconde option car elle peut être calculée avec les informations obtenues lors de l'affinage selon $\mathcal{X}_j$. Ils utilisent des compteurs comprenant le nombre d'images de chaque élément dans les différentes classes. Ces compteurs sont mis à jour lors des étapes d'affinage (pour plus de détails sur ce point, voir [PT87]).

Une conséquence très importante de ce nouveau point concerne la nature des éléments de *Pivots*. Si, comme dans le cas du calcul de la partition fonctionnelle maximum, *Pivots* contient $\mathcal{X}_j$, il sera difficile de retrouver l'ensemble $\mathcal{C} \setminus \mathcal{X}_j$. Donc *Pivots* contiendra des ensembles $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$ tels que la partition courante soit stable pour $R^{-1}(\mathcal{X}_i \cup \ldots \cup \mathcal{X}_k)$. Décrivons la structure de données mise au point par Paigue et Tarjan pour mettre à jour *Pivots*. Elle est décrite à la figure 7.5 et fait appel à une partition $\mathcal{P}'$ des classes de $\mathcal{P}$:

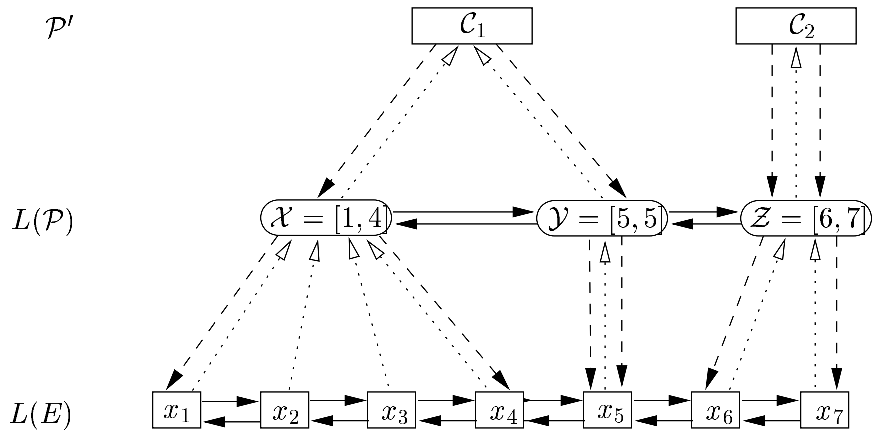


FIG. 7.5 – *Gestion des pivots avec une partition $\mathcal{P}'$ plus grossière que $\mathcal{P}$*

– $\mathcal{P}$ est la partition de E que nous utilisions déjà dans les applications précédentes. C'est la partition courante, elle nous fournira le résultat. Elle est contenue dans une liste doublement chaînée, $L(\mathcal{P})$. Dans la suite, $\mathcal{P}$ sera confondue avec $L(\mathcal{P})$. L'ensembles des classes de $L(\mathcal{P})$ étant partitionné, chaque classe de $L(\mathcal{P})$ possède un pointeur vers sa "super classe" de $\mathcal{P}'$.

– $\mathcal{P}'$ est donc une partition de E plus grossière que $\mathcal{P}$. Soit une classe $\mathcal{C} \in \mathcal{P}'$. Les éléments de chaque classe $\mathcal{C}$ sont consécutifs dans $L(\mathcal{P})$. Et $\mathcal{C}$ possède un pointeur sur le dernier et le premier élément de $L(\mathcal{P})$ qu'elle contient.

  La partition $\mathcal{P}'$ permet d'insérer facilement les nouveaux éléments dans *Pivots*. Supposons que $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$, où $i \leqslant j \leqslant k$, $\mathcal{X}_j$ est une classe de $L(\mathcal{P})$. Alors la partition $L(\mathcal{P})$ est toujours stable pour $R^{-1}(\mathcal{X}_i \cup \ldots \cup \mathcal{X}_k)$.

C'est donc la partition $\mathcal{P}'$ qui fournit les éléments à *Pivots*. Décrivons la manière de gérer ces deux structures. Soit $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$ une classe de $\mathcal{P}'$. Nous savons que $L(\mathcal{P})$ est stable pour $R^{-1}(\mathcal{X}_i \cup \ldots \cup \mathcal{X}_k)$, mais rien ne garantit la stabilité de $L(\mathcal{P})$ selon $R^{-1}(\mathcal{X}_j)$ pour $i \leqslant j \leqslant k$.

Nous sommes donc susceptibles de choisir une sous-classe $\mathcal{Z} = \mathcal{X}_j$ pour $i \leqslant j \leqslant k$ pour affiner $L(\mathcal{P})$ selon $R^{-1}(\mathcal{Z})$ puis $R^{-1}(\mathcal{C} \setminus \mathcal{Z})$. Alors $\mathcal{Z}$ est supprimée de $\mathcal{C}$ et une nouvelle classe $\mathcal{C}_z$ est crée dans $\mathcal{P}'$. La classe $\mathcal{P}'$ qui correspondait à $\mathcal{C}$ correspond désormais à $\mathcal{C} \setminus \mathcal{Z}$.

Si $\mathcal{C} \setminus \mathcal{Z}$ ne contient qu'une seule classe $\mathcal{Y}$ de $\mathcal{P}$, alors $\mathcal{P}$ est stable pour $R^{-1}(\mathcal{Y})$ (car nous avons affiné selon $R^{-1}(\mathcal{C} \setminus \mathcal{Z})$). Dans le cas contraire, rien ne garantit la stabilité des classes de $\mathcal{P}$ incluses dans $\mathcal{C} \setminus \mathcal{Z}$. Nous ajoutons donc le pointeur sur $\mathcal{C} \setminus \mathcal{Z}$ (qui est une classe de $\mathcal{P}'$) à *Pivots*. La partition $\mathcal{P}$ étant stable pour $R^{-1}(\mathcal{Z})$, la classe $\mathcal{C}_z$ de $\mathcal{P}'$ ne sera ajoutée à *Pivots* que lorsque $\mathcal{Z}$ sera affinée.

Pour résumer nous pouvons considérer que *Pivots* contient deux types d'éléments :

1. $\mathcal{E} \in Pivots$ est une ancienne classe de $\mathcal{P}$ qui a été affinée depuis sa dernière utilisation dans une étape d'affinage selon $R^{-1}(\mathcal{E})$ et $R^{-1}(\mathcal{Y} \setminus \mathcal{E})$ où $\mathcal{Y}$ est une union de classes de $\mathcal{P}$.

2. $\mathcal{E} \in Pivots$ est une union de classes de $\mathcal{P}$ et $\mathcal{P}$ a été affinée par $R^{-1}(\mathcal{E})$ grâce à une classe $\mathcal{X}$.

De manière générale, nous pouvons ajouter une classe $\mathcal{C}$ de $\mathcal{P}'$ à *Pivots* dès qu'elle ne correspond plus à une unique classe de $L(\mathcal{P})$. Ce test peut facilement être effectué à chaque fois qu'une classe de $L(\mathcal{P})$ est scindée en utilisant la structure de données de la figure 7.5. Il suffit de comparer les pointeurs sur le premier et le dernier élement de $\mathcal{C}$.

L'algorithme principal aussi doit être réécrit. La partition courante étant stable pour chaque classe pivot $\mathcal{C}$ choisie, la règle de Hopcroft peut être mise en oeuvre en choisissant une classe $\mathcal{Z}$ de $L(\mathcal{P})$ incluse dans $\mathcal{C}$ telle que $|\mathcal{Z}| \leqslant (\sum_{i \leqslant j \leqslant k} |\mathcal{X}_j|)/2$.

Algorithme 20: Affinage de Partition

Données : une partition $L(\mathcal{P}) = (\mathcal{X}_1, \ldots, \mathcal{X}_r)$ d'un ensemble E en classes et une partition $\mathcal{P}'$ des classes de $L(\mathcal{P})$

Résultat : un affinage de la partition $L(\mathcal{P})$

début

  $Pivots \leftarrow \emptyset$;

  tant que *la procédure InitPartition n'arrête pas la boucle* **faire**

    InitPartition $(L(\mathcal{P}))$;

    tant que $Pivots \neq \emptyset$ **faire**

      choisir une classe pivot $\mathcal{C}$ dans $Pivots$;

      soit $\mathcal{Z}$ est classe de $L(\mathcal{P})$ incluse dans $\mathcal{C}$ telle que $|\mathcal{Z}| \leqslant |\mathcal{C}|/2$;

      remplacer $\mathcal{C}$ dans $\mathcal{P}'$ par $\mathcal{Z}, \mathcal{C} \setminus \mathcal{Z}$;

      si $\mathcal{C} \setminus \mathcal{Z}$ *contient plusieurs classes de* $L(\mathcal{P})$ **alors**

        ajouter $\mathcal{C} \setminus \mathcal{Z}$ à $Pivots$;

      ClassePivot $(\mathcal{Z})$;

fin

Avant de présenter l'algorithme de Paigue et Tarjan dans notre formalisme, nous illustrons son principe avec la figure 7.6, en insistant sur la mise à jour de $Pivots$ ($Pivots$ est initialisé avec l'ensemble E) :

- A la première étape, le seul élément de $Pivots$ est E. Nous choisissons donc une classe $\mathcal{Z} \subset E$ telle que $|\mathcal{Z}| \leqslant |E|/2$: prenons $\mathcal{X}$. Alors la classe E est remplacée par $\mathcal{X}$ et $E \setminus \mathcal{X}$ dans $\mathcal{P}'$. Puisque $\mathcal{E} \setminus \mathcal{X} = \mathcal{Y}$, cette classe n'est pas ajoutée à $Pivots$. Nous devons faire successivement les opérations : *Affiner*$(\mathcal{P}, R^{-1}(\mathcal{X}))$ et *Affiner*$(\mathcal{P}_1, R^{-1}(E \setminus \mathcal{X}))$.

- A l'issue de $\mathcal{P}_1 =$ *Affiner*$(\mathcal{P}, R^{-1}(\mathcal{X}))$, les classes $\mathcal{X}$ et $\mathcal{Y}$ ont été affinées. Nous devons donc les ajouter à $Pivots$.

- A l'issue de $\mathcal{P}_2 =$ *Affiner*$(\mathcal{P}_1, R^{-1}(E \setminus \mathcal{X})$, $Pivots$ ne change pas.

- A la seconde étape, nous pouvons choisir $\mathcal{Y}$ dans $Pivots$. La classe $\mathcal{Y}''_a$ vérifie $|\mathcal{Y}''_a| \leqslant |\mathcal{Y}|/2$. Nous pouvons donc la choisir pour affiner $\mathcal{P}_2$. La partition $\mathcal{P}'$ devient alors $\mathcal{P}' = \{\mathcal{X}, \mathcal{Y}''_a, \mathcal{Y} \setminus \mathcal{Y}''_a\}$. Puisque $\mathcal{Y} \setminus \mathcal{Y}''_a$ n'est pas restreint à une seule classe, nous l'ajoutons à $Pivots$, donc $Pivots = \{\mathcal{X}, \mathcal{Y} \setminus \mathcal{Y}''_a\}$. Nous devons faire successivement les opérations : *Affiner*$(\mathcal{P}, R^{-1}(\mathcal{Y}''_a))$ et *Affiner*$(\mathcal{P}_1, R^{-1}(Y \setminus \mathcal{Y}''_a))$.

- Lors de l'opération $\mathcal{P}_3 =$ *Affiner*$(\mathcal{P}_2, R^{-1}(\mathcal{Y}''_a))$, la classe $\mathcal{Y}''_a$ est affinée. Nous devons donc aussi l'ajouter à $Pivots$: $Pivots = \{\mathcal{X}, \mathcal{Y} \setminus \mathcal{Y}''_a, \mathcal{Y}''_a\}$.

- Normalement, il faut alors effectuer l'opération $\mathcal{P}_4 = Affiner(\mathcal{P}_3, R^{-1}(\mathcal{Y} \setminus \mathcal{Y}_a''))$. Mais à ce stade, la partition ne contient que des singletons. Nous pouvons donc arrêter la description.

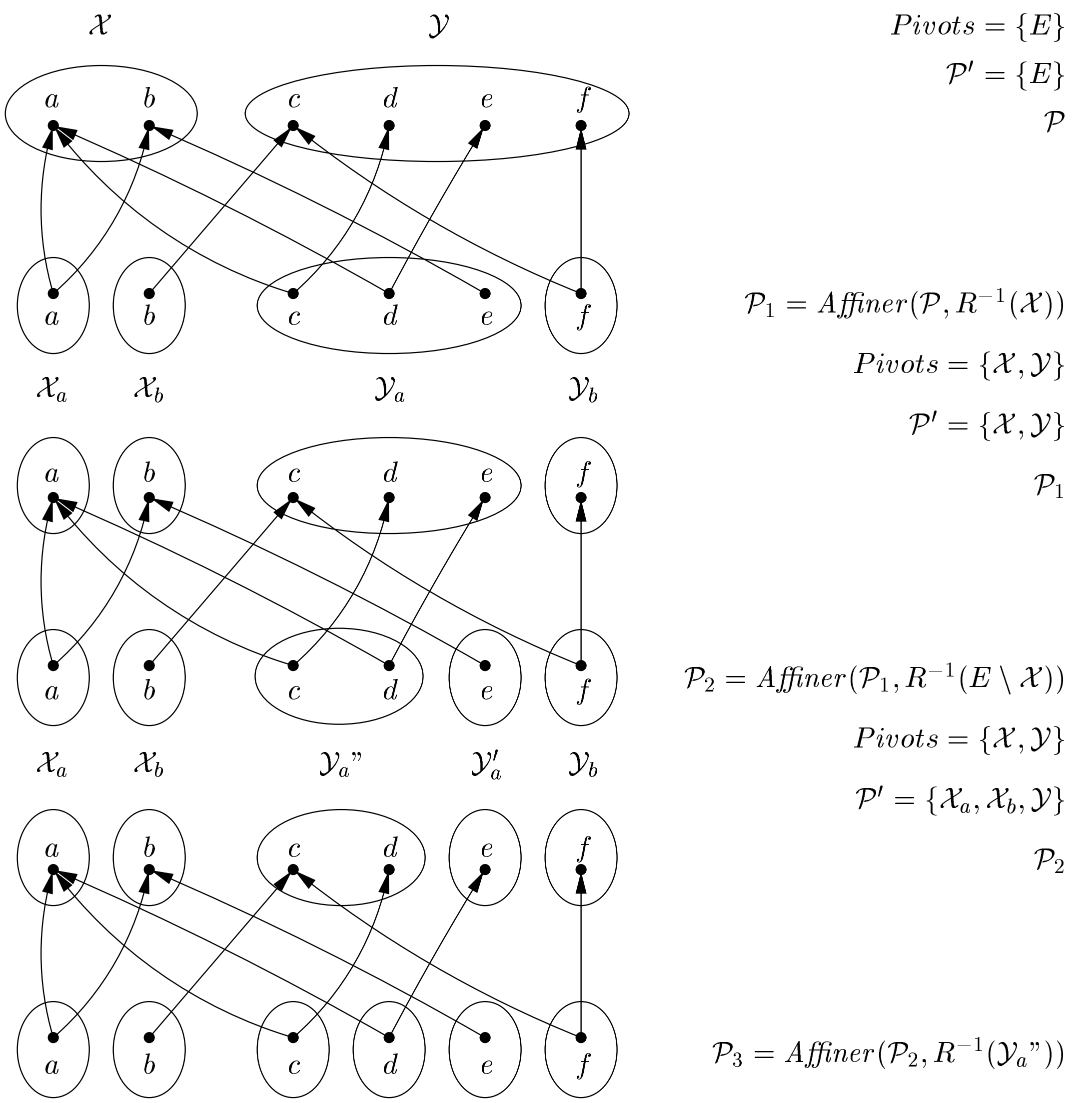


FIG. 7.6 – *Calcul de la partition relationnelle maximum de $\mathcal{P} = \{\mathcal{X}.\mathcal{Y}\}$.*

La preuve de l'algorithme 21 ressemble beaucoup à celle du calcul de la partition fonctionnelle maximum. La seule différence réside dans la non-validité du lemme 7.4 pour la partition relationnelle maximum.

Algorithme 21: Partition relationnelle maximum

Données : une relation binaire R sur E telle que $|R(x)| \geqslant 1$ pour tout $x \in E$ et une partition $\mathcal{Q}$

Résultat : la partition relationnelle $L(\mathcal{P})$ pour la partition $\mathcal{Q}$ et la relation R

début

  $L(\mathcal{P}) \leftarrow \mathcal{Q}$;

  $\mathcal{P}' \leftarrow (L(\mathcal{P}))$;

  exécuter l'algorithme 20 et les procédures 11, 19 avec les définitions suivantes (*Pivots* contient les classes non-singletons de $\mathcal{P}'$);

fin

Procedure *InitPartition* $(L(\mathcal{P}))$

  au premier appel :

    choisir une classe $\mathcal{Z}$ telle que $|\mathcal{Z}| \leqslant |E|/2$;

    remplacer E par $\mathcal{Z}, E \setminus \{\mathcal{Z}\}$ dans $\mathcal{P}'$;

    si *$E \setminus \{\mathcal{Z}\}$ est constitué de plusieurs classes de $L(\mathcal{P})$* **alors**

      ajouter $E \setminus \mathcal{Z}$ à *Pivots*;

  au second appel :

    fin du processus, **retourner** $L(\mathcal{P})$;

Procedure *EnsemblePivot* $(p = (x, \mathcal{X}))$

  retourner $R^{-1}(x)$;

Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$

  retourner vrai;

Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_a)$

  si *$\mathcal{X}$ et $\mathcal{X}_a$ sont les seuls éléments d'une classe $\mathcal{C}$ de $\mathcal{P}'$* **alors**

    ajouter $\mathcal{C}$ à *Pivots*;

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Toute partition relationnelle pour la partition $\mathcal{Q}$ et la relation R est plus fine que $L(\mathcal{P})$; $L(\mathcal{P})$ est compatible avec $\mathcal{Q}$*
- ***B:*** *$L(\mathcal{P})$ est stable pour $R^{-1}(\mathcal{X})$ pour toute classe $\mathcal{X} \in L(\mathcal{P})$ $(L(\mathcal{P}) = \mathcal{P}')$*

Preuve : Grâce à une induction nous pouvons que chaque étape d'affinage préserve la propriété A. Cette propriété est vraie après le premier appel à la procédure

InitPartition car $L(\mathcal{P}) = \mathcal{Q}$, donc $L(\mathcal{P})$ compatible avec $\mathcal{Q}$ et toute partition relationnelle pour $\mathcal{Q}$ est plus fine que $L(\mathcal{P})$. Supposons que la propriété soit vraie avant chaque étape d'affinage.

Soient $L(\mathcal{P})$ la partition courante et $\mathcal{C}$ un élément de *Pivots*. Alors $\mathcal{C}$ contient plusieurs classes de $L(\mathcal{P})$. Soit $\mathcal{Z}$ la classe de $L(\mathcal{P})$ choisie. Nous allons successivement faire les opérations $\mathcal{P}_1 = \mathit{Affiner}(L(\mathcal{P}), R^{-1}(\mathcal{Z})$ et $\mathcal{P}_2 = \mathit{Affiner}(\mathcal{P}_1, R^{-1}(\mathcal{C} \setminus \mathcal{Z})$. Tout d'abord, la structure de données utilisée garantit que $\mathcal{P}_1$ soit compatible avec $L(\mathcal{P})$ et $\mathcal{P}_{\in}$ avec $\mathcal{P}_1$. Donc par induction $\mathcal{P}'$ est compatible avec $\mathcal{Q}$.

Par hypothèse d'induction toute partition relationnelle $\mathcal{R}$ est plus fine que $L(\mathcal{P})$. Montrons que $\mathcal{R}$ est plus fine que $\mathcal{P}_1$. Supposons qu'il existe une classe $\mathcal{X}^r \in \mathcal{R}$ qui ne soit incluse dans aucune classe de $\mathcal{P}_1$. Puisque $\mathcal{R}$ est plus fine que $L(\mathcal{P})$, la classe $\mathcal{X}^p \in L(\mathcal{P})$ qui contenait $\mathcal{X}^r$ a été coupée en deux sous-classes $\mathcal{X}^p_a = \mathcal{X}^p \cap R^{-1}(\mathcal{Z})$ et $\mathcal{X}^p_b = \mathcal{X}_p \setminus f^{-1}(\mathcal{Z})$. Puisque X^r n'est inclus ni dans $\mathcal{X}^p_a$ ni dans $\mathcal{X}^p_b$, $\mathcal{X}^r$ contient des éléments de ces deux classes. Autrement dit, $\mathcal{X}^r$ n'est pas stable pour $f^1(\mathcal{Z})$, donc $\mathcal{R}$ n'est pas une partition fonctionnelle : contradiction, $\mathcal{R}$ est plus fine que $\mathcal{P}_1$. La même preuve peut être faite pour $\mathcal{P}_2$.

Montrons que le processus termine avec le résultat souhaité : *i.e.* lorsque $\mathit{Pivots} = \emptyset$ alors la propriété B est vérifiée. Soit $\mathcal{C}$ une classe de $\mathcal{P}'$. Puisque $\mathit{Pivots}\emptyset$, $\mathcal{C}$ ne contient qu'une classe $\mathcal{X}$ de $L(\mathcal{P})$. Il y a deux situations possibles :

1. $\mathcal{X}$ a été choisie pour affiner la partition $L(\mathcal{P})$: une nouvelle classe ne contenant que $\mathcal{X}$ a été créée dans $\mathcal{P}'$. Puisque cette classe n'est pas dans *Pivots*, $\mathcal{X}$ n'a jamais été coupée. Donc $L(\mathcal{P})$ est stable pour $R^{-1}(\mathcal{X})$.

2. $\mathcal{X}$ n'a pas été choisie pour affiner la partition $L(\mathcal{P})$: soit $\mathcal{C}'$ la dernière classe de $\mathcal{P}'$ qui contenait $\mathcal{C}$. Alors $\mathcal{C}'$ ne contenait que deux classes $\mathcal{X}$ et la classe $\mathcal{Y}$ qui a été utilisée pour affiner $L(\mathcal{P})$. $\mathcal{C}$ n'a donc pas été ajoutée à *Pivots* et depuis $\mathcal{X}$ n'a jamais été coupée. Lors de l'étape d'affinage avec la classe $\mathcal{Y}$, nous avons aussi pu affiner $L(\mathcal{P})$ par $R^{-1}(\mathcal{C}' \setminus \mathcal{Y})$. Or $\mathcal{C}' \setminus \mathcal{Y} = \mathcal{X}$ donc $L(\mathcal{P})$ est stable pour $R^{-1}(\mathcal{X})$.

Puisque le processus s'arrête lorsque *Pivots* est vide, cela signifie que $L(\mathcal{P})$ est stable pour $R^{-1}(\mathcal{X})$ quel que soit $\mathcal{X} \in L(\mathcal{P})$: $L(\mathcal{P})$ est donc une partition relationnelle pour $\mathcal{Q}$ et la relation R.

De plus, la propriété A montre que $L(\mathcal{P})$ est plus grossière que toutes les partitions relationnelles pour $\mathcal{Q}$. Puisque c'est elle-même une partition relationnelle, c'est l'unique partition relationnelle maximum (*i.e.* avec la moins de classes). Le lemme 7.3 est donc prouvé. Puisque A montre aussi que $L(\mathcal{P})$ est compatible avec $\mathcal{Q}$, alors $L(\mathcal{P})$ est une partition relationnelle maximum pour la relation R compatible avec $\mathcal{Q}$. □

Analyse de complexité : N'ayant pas détaillé entièrement les structures de données utilisées par Paigue et Tarjan (notamment celles utilisées pour l'étape d'affinage), nous n'analyserons pas sa complexité. Toutefois si nous admettons que l'affinage par $\mathcal{Z}$ et $\mathcal{C} \setminus \mathcal{Z}$ peut se faire en $O(|S|)$ où $S = R^{-1}(\mathcal{Z})$, alors nous pouvons nous convaincre d'une complexité en $O(m \log n)$ avec $m = \Sigma_{x \in E}|R(x)|$. Le facteur $\log n$ provient de l'utilisation de la règle de Hopcroft : quand une classe $\mathcal{X}$ de $\mathcal{P}$ est choisie parmi les classes incluses dans $\mathcal{C} = \mathcal{X}_i \cup \ldots \cup \mathcal{X}_k$, alors nous assurons que $|\mathcal{Z}| \leqslant (\Sigma_{i \leqslant j \leqslant k}|\mathcal{X}_j|)/2$. Ainsi l'ensemble $R^{-1}(x)$ pour chaque élément $x \in E$ est utilisé au plus $\log n$ dans un ensemble pivot. $\square$

7.4 Conclusion

La règle de Hopcroft s'avère donc un outil puissant et général pour obtenir des algorithmes efficaces ($O(n + m \log n)$). Avec les trois applications présentées, nous avons pu étudier deux manières différentes d'appliquer cette règle. La première repose sur des propriétés algébriques du problème. La seconde, au contraire, n'est possible que grâce à une étude approfondie des structures de données nécessaires.

Comme nous allons le voir aux chapitres suivants, l'utilisation de cette règle dénote parfois d'un manque d'informations pour choisir systématiquement un pivot qui affine strictement la partition courante. Ces informations peuvent être très difficiles à obtenir et à gérer efficacement. Ainsi, pour certaines applications, nous aurons le choix entre un algorithme linéaire très compliqué et un algorithme simple utilisant la règle de Hopcroft. C'est en particulier le cas de la décomposition modulaire et de l'orientation transitive que nous étudions au chapitre suivant.

Chapitre 8

Règle de Hopcroft appliquée aux graphes

Les applications que nous présentons dans ce chapitre utilisent toutes la notion de *module* d'un graphe. Nous commençons donc par faire quelques brefs rappels sur la décomposition modulaire. Nous nous inspirons des applications du chapitre 7 (en particulier de la partition fonctionnelle maximum) pour proposer un algorithme qui, étant donné une partition des sommets d'un graphe, calcule la partition modulaire maximum. Nous mettons ainsi en évidence les différences entre la décomposition modulaire et les applications du chapitre 7. L'algorithme proposé peut servir de base à un algorithme complet de décomposition modulaire.

Nous étudions ensuite un cas particulier de décomposition modulaire : la décomposition série/parallèle. Elle caractérise les cographes. Un nouvel algorithme de reconnaissance de cette classe de graphe est présenté. Bien que sa complexité ne soit pas optimale, cet algorithme nous semble intéressant du point de vue affinage de partition. En effet, à certaines étapes, nous disposons d'informations suffisantes pour obtenir efficacement un pivot qui affine strictement la partition; à d'autres étapes nous devrons avoir recours à la règle de Hopcroft.

Enfin, nous abordons le problème de l'orientation transitive d'un graphe et montrons les liens forts entre l'orientation transitive d'un graphe et la décomposition modulaire [MS94]. L'algorithme d'orientation présenté généralise aux graphes non-premiers les algorithmes connus. Cette généralisation est rendue possible par l'utilisation de Lex-BFS.

Il existe déjà des algorithmes linéaires pour résoudre ces trois problèmes. Mais l'utilisation des techniques d'affinage de partition produit des algorithmes plus simples à comprendre et à implémenter.

8.1 Décomposition modulaire

L'idée naturelle de toute décomposition de graphe est de regrouper des sommets (ou sous-graphes) qui possèdent des propriétés similaires. Pour ce qui nous concerne, nous nous intéressons aux modules d'un graphe :

Définition 8.1 *Un* ***module*** *d'un graphe $G = (V, E)$ est un ensemble de sommets $S \subseteq V$ tel que pour tout $x, y \in M$, nous avons $N(x) \setminus M = N(y) \setminus M$.*

Autrement dit, un module est un ensemble de sommets M ayant le même voisinage à l'extérieur de M. Clairement l'ensemble vide, l'ensemble des sommets d'un graphe et tous les singletons constitués d'un sommet du graphe sont des modules. Nous dirons que ce sont les *modules triviaux.*

Définition 8.2 *Un graphe est dit* ***premier****, ou indécomposable, s'il ne contient que des modules triviaux.*

La décomposition modulaire consiste donc à calculer les modules d'un graphe. Lorsqu'un module M est calculé, alors un sommet x_m est substitué à M ($N(x_m) = N(x) \setminus M$ pour $x \in M$). Nous dirons aussi que M est contracté en un sommet x_m.

Le nombre de modules d'un graphe peut être important (tout sous-ensemble du graphe complet est un module), nous ne nous intéressons qu'aux *modules forts* d'un graphe :

Définition 8.3 *Un module M du graphe $G = (V, E)$ est un* ***module fort*** *si et seulement si pour tout module M', $M' \cap M \neq \emptyset$ implique $M \subseteq M'$ ou $M' \subseteq M$.*

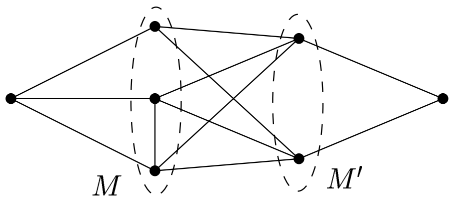


FIG. 8.1 – *M et M' sont des modules*

Du point de vue de l'affinage de partition, les modules forts sont intéressants car ils vérifient la propriété suivante :

Lemme 8.4 *Les modules forts maximaux (pour l'ordre d'inclusion) d'un graphe $G = (V, E)$ différents de V induisent une partition de l'ensemble des sommets de G. Cette partition est appelée* ***décomposition en modules*** *de G.*

8.1.1 Les différentes décompositions

Puisque nous nous plaçons dans le cas des graphes non-orientés, il existe trois types de décomposition par substitution d'un graphe.

Décomposition parallèle

Chaque composante connexe d'un graphe non-connexe $G = (V, E)$ est un module. L'ensemble des composantes connexes de G induit une partition en module de G. Le graphe associé est un stable.

Décomposition série

De même, une composante connexe du complémentaire d'un graphe connexe $G = (V, E)$ est un module. L'ensemble des composantes connexes de $\overline{G}$ induit une partition en modules de G. Le graphe associé est un graphe complet.

Décomposition premier

En revanche, si G et $\overline{G}$ sont connexes, alors il existe un sous-ensemble de sommets $Y \subset V$ tel que $|Y| > 2$ et une unique partition $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_k\}$ de V, tels que :

1. $G[Y]$ est un sous-graphe premier,
2. Pour tout i, $1 \geqslant i \geqslant k$, $\mathcal{X}_i$ est un module de G,
3. Pour tout i, $1 \geqslant i \geqslant k$, $|\mathcal{X}_i \cap Y| = 1$.

8.1.2 Théorème de décomposition

Le théorème suivant montre l'unicité de la décomposition en modules d'un graphe. Nous ne présentons que la version des graphes non-orientés. Dans le cas de graphes orientés, il faut rajouter un quatrième cas. Différentes versions de ce théorème sont apparues dans la littérature. Nous pouvons citer les travaux de Galaï [Gal67], de M. Habib [Hab81], de U. Faigle, O. Goecke et R. Schrader [FGS].

Théorème 8.5 *Soit $G = (V, E)$ un graphe tel que $|V| \geqslant 2$, alors un seul des trois cas suivants est vérifié :*

1. *G est un graphe non-connexe : c'est une décomposition* parallèle*,*
2. *$\overline{G}$ est un graphe non-connexe : c'est une décomposition* série*,*
3. *G et $\overline{G}$ sont des graphes connexes : c'est une décomposition* premier.

Si nous nous intéressons récursivement à chacun des sous-graphes induits par les modules forts d'un graphes nous obtenons un *arbre de décomposition* du graphe. Les feuilles de cet arbre sont les sommets du graphe et chaque noeud interne sera étiqueté par le type de la décomposition associée. Le théorème précédent montre l'unicité de l'arbre de décomposition d'un graphe. La figure 8.2 propose un exemple d'arbre de décomposition d'un graphe.

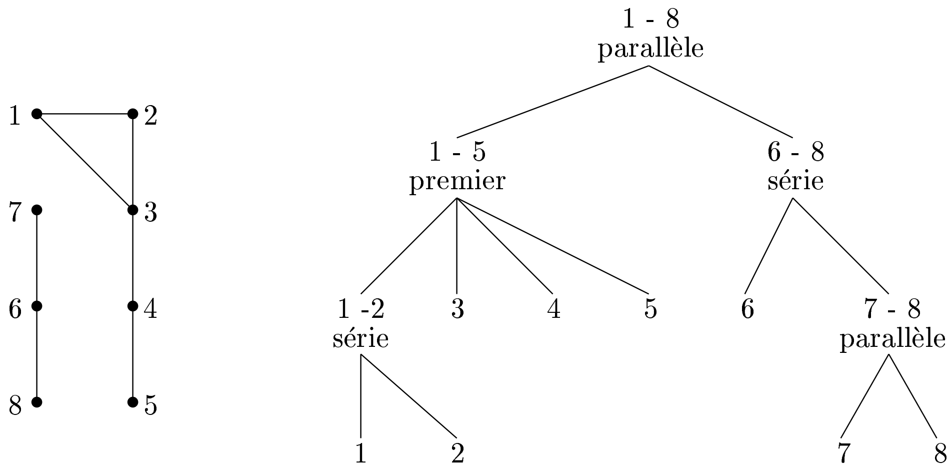


FIG. 8.2 – *Un graphe G et son arbre de décomposition modulaire*

De nombreux travaux ont porté sur le calcul de l'arbre de décomposition d'un graphe. Les premiers algorithmes linéaires sont dûs à Cournier et Habib [CH94] et Mc Connell et Spinrad [MS94]. Plus récemment, C. Capelle [Cap96] a étudié ce problème en introduisant la notion de *permutation factorisante* :

Définition 8.6 *Une* ***permutation factorisante*** *est une permutation σ des sommets d'un graphe $G = (V, E)$ telle que tout module fort un est facteur de σ.*

Autrement dit pour tout module fort M, l'ensemble des sommets de M sont consécutifs dans σ. Etant donné un arbre de décomposition, si nous numérotons ses feuilles de droite à gauche, nous obtenons une permutation factorisante du graphe. Par exemple dans la figure 8.2, la permutation $\sigma = [1, 2, 3, 4, 5, 6, 7, 8]$ est une permutation factorisante.

Théorème 8.7 *[Cap96] Etant donnée un permutation factorisante σ d'un graphe $G = (V, E)$, un arbre de décomposition modulaire peut être calculé en $O(n + m)$.*

8.1.3 Un algorithme

L'algorithme que nous présentons dans cette section peut être vu comme une première étape vers un algorithme de décomposition modulaire. Nous verrons à la fin de cette section, comment il pourrait être étendu. Le même genre d'idées est à la base de l'algorithme en $O(n + m \log n)$ de Mc Connell et Spinrad [MS94]. Il est intéressant mettre en parallèle les problèmes du chapitre 7 (partition fonctionnelle maximum et partition relationnelle maximum) et la décomposition modulaire.

Définition 8.8 *Soit $\mathcal{Q} = \{\mathcal{B}_1, \ldots, \mathcal{B}_q\}$ une partition des sommets d'un graphe $G = (V, E)$. Une* ***partition modulaire*** *pour $\mathcal{Q}$ est une partition $\mathcal{P} = \{\mathcal{X}_1, \ldots, \mathcal{X}_p\}$ telle que :*

1. *$\mathcal{P}$ est plus fine que $\mathcal{Q}$: i.e. $\mathcal{P} \ll \mathcal{Q}$*
2. *chaque classe $\mathcal{X}_i$ de $\mathcal{P}$ est un module de G*

Le problème qui nous intéresse est la calcul de la partition modulaire maximum. Nous montrons que cette partition est unique (voir preuve de l'invariant).

Lemme 8.9 *Il existe une unique partition modulaire maximum pour $\mathcal{Q}$.*

La première différence entre le problème de la partition modulaire maximum et celui de la partition fonctionnelle maximum concerne le calcul des ensembles pivots. Lorsqu'une classe $\mathcal{X}$ est utilisée comme pivot, il faut affiner la partition $\mathcal{P}$ avec chacun des ensembles pivots associés aux sommets de $\mathcal{X}$ (*i.e.* pour $x \in \mathcal{X}$, $N(x) \setminus \mathcal{X}$). Alors toutes les classes de $\mathcal{P}$ sont stables pour les sommets de $\mathcal{X}$. Autrement dit, deux éléments d'une même classe $\mathcal{Y}$ ont le même voisinage dans $\mathcal{X}$. La procédure *ClassePivot* est donc la procédure 10. En ce qui concerne la partition fonctionnelle maximum, un seul ensemble pivot était calculé par classe. Il était obtenu comme l'union des ensembles associés à chacun de ses éléments. Dans la décomposition modulaire, nous pouvons voir le calcul des ensembles pivots comme une "intersection".

Ensuite, contrairement au problème de la partition fonctionnelle maximum, il n'existe pas de propriété similaire au lemme 7.4. En effet, supposons qu'à une étape donnée, la classe $\mathcal{X}$ soit utilisée pour affiner $\mathcal{P}$. Après cette étape, toutes les classes $\mathcal{Y} \neq \mathcal{X}$ de $\mathcal{P}$ sont stables pour les voisinages de tous les sommets de $\mathcal{X}$. Et ce sera le cas pour toutes les futures sous-classes de $\mathcal{Y}$. Nous dirons donc que $\mathcal{P}$ est stable pour $\mathcal{X}$. Supposons maintenant que la classe $\mathcal{X}$ soit elle-même affinée en $\mathcal{X}_a$ et $\mathcal{X}_b$. Une propriété équivalente au lemme 7.4 nous dirait que affiner $\mathcal{P}$ selon les ensembles pivots associés à $\mathcal{X}_a$ ou $\mathcal{X}_b$ produit le même résultat. Ce n'est pas le cas ici pour la simple raison qu'une classe n'est pas affinée par les ensembles pivots qu'elle génère (voir figure 8.3).

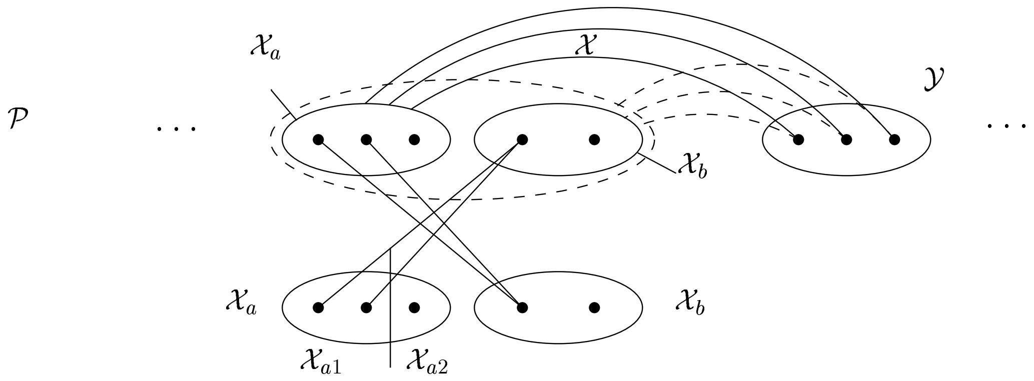


FIG. 8.3 – *La partition $\mathcal{P}$ est stable pour $\mathcal{X}$. La classe $\mathcal{Y}$ a affinée $\mathcal{X}$ en $\mathcal{X}_a$ et $\mathcal{X}_b$. Lorsque l'on affine $\mathcal{P}$ avec $\mathcal{X}_b$, la classe $\mathcal{X}_a$ est affinée. A ce stade la partition n'est pas stable selon $\mathcal{X}_a$. C'est en particulier le cas pour $\mathcal{X}_b$*

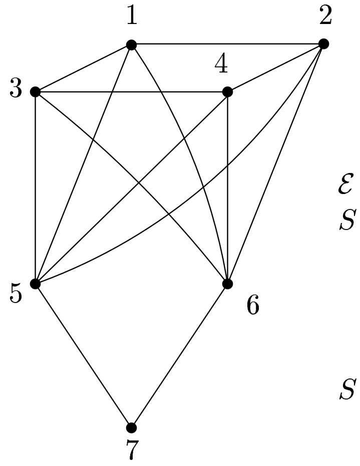
Nous nous retrouvons donc dans une situation similaire au problème de la partition relationnelle maximum. Dans notre cas aussi, nous pouvons résoudre facilement le problème grâce à la propriété suivante :

Lemme 8.10 *Soit $\mathcal{P} = \mathcal{X}_1 \cup \ldots \cup \mathcal{X}_k$ une partition des sommets d'un graphe G stable pour tout $\mathcal{X}_i$ où $2 \leqslant i \leqslant k$, alors $\mathcal{X}_1$ est un module.*

Preuve : Puisque $\mathcal{P}$ est stable pour tout $\mathcal{X}_i$ où $2 \leqslant i \leqslant k$, alors $\mathcal{X}_1$ en particulier l'est aussi. Cela signifie que, pour tout sommet $x \notin \mathcal{X}_1$, $\mathcal{X}_1$ est stable pour $N(x)$. Donc tous les sommets de $\mathcal{X}_1$ ont le même voisinage à l'extérieur de $\mathcal{X}_1$. C'est donc un module. □

Dans la situation du lemme 8.10, rien ne garantit que la partition soit stable pour les ensembles pivots associés à $\mathcal{X}_1$. Mais $\mathcal{X}_1$ est un module : d'une part cette classe ne pourra plus être coupée, d'autre part tous ses sommets ont le même voisinage à l'extérieur. Nous pouvons donc choisir un sommet arbitraire dans $\mathcal{X}_1$ pour affiner la partition $\mathcal{P}$. Cette situation arrivera au plus une fois par classe, elle n'entrainera donc pas de surcoût de complexité.

Pour gérer les classes telles que $\mathcal{X}_1$, une file, nommée $Modules$, est utilisée. Elle contient des pointeurs sur des classes. Chaque classe possède aussi un pointeur vers son élément correspondant. Ainsi une classe peut-être rajoutée et supprimée en temps constant.



$\mathcal{P} = \{\{1,2,4,6\},\{3,5,7\}\}$

Pivots : $\{3,5,7\}$ — Modules : $\{1,2,4,6\}$

$\mathcal{E} = \{3,5,7\}$:
$S = N(3) \setminus \mathcal{E} = \{1,4,6\}$ — $\mathcal{P}_1 = \{\{2\},\{1,4,6\},\{3,5,7\}\}$

Pivots : $\{2\}$ — Modules : $\{1,4,6\}$

$S = N(5) \setminus \mathcal{E} = \{1,2,4\}$ — $\mathcal{P}_2 = \{\{2\},\{6\},\{1,4\},\{3,5,7\}\}$

Pivots : $\{6\}$, $\{2\}$ — Modules : $\{1,4\}$

$S = N(7) \setminus \mathcal{E} = \{6\}$: ne fait rien

$\mathcal{E} = \{2\}$:
$S = N(2) \setminus \mathcal{E} = \{1,4,5,6\}$ — $\mathcal{P}_3 = \{\{2\},\{6\},\{1,4\},\{3,7\},\{5\}\}$

Pivots : $\{5\}$, $\{6\}$ — Modules : $\{3,7\}$, $\{1,4\}$

$\mathcal{E} = \{6\}$:
$S = N(6) \setminus \mathcal{E} = \{1,2,3,4,7\}$: ne fait rien

$\mathcal{E} = \{5\}$:
$S = N(5) \setminus \mathcal{E} = \{1,2,3,4,7\}$: ne fait rien

Pivots : (vide) — Modules : $\{3,7\}$, $\{1,4\}$

$\mathcal{E} = \{1,4\}$: il suffit de n'utiliser qu'un sommet
$S = N(1) \setminus \mathcal{E} = \{2,3,5,6\}$: — $\mathcal{P}_4 = \{\{2\},\{6\},\{1,4\},\{7\},\{3\},\{5\}\}$

Pivots : $\{3\}$ — Modules : $\{7\}$

La partition est stable.

FIG. 8.4 – *Partition modulaire maximum pour* $\mathcal{P} = \{\{1,2,4,6\},\{3,5,7\}\}$.

La figure 8.4 décrit la manière de gérer *Pivots* et *Modules*. Puisque l'ensemble des sommets est un module, *Modules* est initialisé avec la plus grande classe de $\mathcal{P}$. Les autres sont ajoutées à *Pivots*. Soit $\mathcal{X}$ une classe affinée en $\mathcal{X}_a$ et $\mathcal{X}_b$:

- Si $\mathcal{X} \in Pivots$, alors la partition n'est pas stable pour les éléments de $\mathcal{X}$. Donc X_a est ajoutée à *Pivots* ($\mathcal{X}_b$ y est toujours).

– Sinon, que $\mathcal{X}$ appartienne ou non à $Modules$, le lemme 8.10 s'applique : la plus grande sous classe est ajoutée à $Modules$, la plus petite à $Pivots$.

Algorithme 22: Calcul d'une partition modulaire maximum

Données : Un graphe $G = (V, E)$ et une partition $\mathcal{Q}$ des sommets V
Résultat : Une partition $\mathcal{P}$ en modules maximaux compatible avec $\mathcal{Q}$
début
  $\mathcal{P} \leftarrow (V)$;
  exécuter l'algorithme 9 et la procédure 10 avec les définitions suivantes (les éléments de $Pivots$ sont des classes de $\mathcal{P}$);
fin
Procedure *InitPartition* $(\mathcal{P})$
  au premier appel :
    $\mathcal{P} \leftarrow \mathcal{Q}$;
    ajouter la plus grande classe de $\mathcal{P}$ à $Modules$;
    ajouter les autres classes de $\mathcal{P}$ à $Pivots$;
  aux appels suivants :
    si $Modules = \emptyset$ **alors** fin du processus, **retourner** $\mathcal{P}$;
    sinon
      choisir la première classe $\mathcal{X}$ de $Modules$;
      choisir un sommet $x \in \mathcal{X}$;
      ajouter $\{x\}$ à $Pivots$;
Procedure *EnsemblePivot* $(p = (x, \mathcal{X}))$
  retourner $N(x) \setminus \mathcal{X}$;
Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$
  retourner vrai;
Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_a)$
  soit $\mathcal{Z}$ (*resp.* $\mathcal{Z}'$) la plus petite (*resp.* grande) classe parmi $\mathcal{X}$ et $\mathcal{X}_a$;
  si $\mathcal{X} \in Pivots$ **alors** ajouter $\mathcal{X}_a$ à $Pivots$;
  sinon
    si $\mathcal{X} \in Modules$ **alors** échanger $\mathcal{X}$ par $\mathcal{Z}'$ dans $Modules$;
    sinon ajouter $\mathcal{Z}'$ à $Modules$;
    ajouter $\mathcal{Z}$ à $Pivots$;

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots ou Modules qui affine strictement $\mathcal{P}$.*

- ***A:*** *Toute partition modulaire pour $\mathcal{Q}$ est plus fine que $\mathcal{P}$; $\mathcal{P}$ est compatible avec $\mathcal{Q}$.*
- ***B:*** *$\mathcal{P}$ est stable pour tout ensemble $N(x) \setminus \mathcal{X}$ où x est un sommet de $\mathcal{X} \in \mathcal{P}$.*

Preuve : Nous montrons par induction que la propriété A est préservée lors de chaque étape d'affinage. Elle l'est après l'exécution de la procédure *InitPartition* car $\mathcal{P} = \mathcal{Q}$. Supposons qu'elle soit vérifiée avant chaque étape d'affinage.

Soit $\mathcal{P}$ la partition courante et $\mathcal{X} \in Pivots$ telle que $\mathcal{P}$ ne soit pas stable pour $S = N(x) \setminus \mathcal{X}$ où $x \in \mathcal{X}$. Notons $\mathcal{P}' = \mathit{Affiner}(\mathcal{P}, S)$.

Par hypothèse d'induction, toute partition fonctionnelle $\mathcal{R}$ est plus fine que $\mathcal{P}$. Montrons que $\mathcal{R}$ est plus fine que $\mathcal{P}'$. Supposons qu'il existe une classe $\mathcal{X}^r \in \mathcal{R}$ qui ne soit incluse dans aucune classe de $\mathcal{P}'$. Puisque $\mathcal{R}$ est plus fine que $\mathcal{P}$, la classe $\mathcal{X}^p \in \mathcal{P}$ qui contenait $\mathcal{X}^r$ a été coupée en deux sous-classes $\mathcal{X}^p_a = \mathcal{X}^p \cap S$ et $\mathcal{X}^p_b = \mathcal{X}^p \setminus S$. Puisque X^r n'est inclus ni dans $\mathcal{X}^p_a$ ni dans $\mathcal{X}^p_b$, $\mathcal{X}^r$ contient des éléments de ces deux classes, donc des voisins et des non-voisins de x. Autrement dit, $\mathcal{X}^r$ n'est pas stable pour $N(x)\setminus\mathcal{X}$, donc $\mathcal{R}$ n'est pas une partition fonctionnelle : contradiction.

Supposons que *Pivots* soit vide. Alors la partition courante a été affinée par les ensembles pivots associés à toutes les classes qui ne sont pas dans *Modules*. La propriété A est vérifiée. Soit $\mathcal{X}$ la première classe ajoutée à *Modules* (c'est sans doute une sous-classe de la première classe ajoutée, mais elles ont été échangées). Montrons que $\mathcal{X}$ est un module.

Pour chaque classe $\mathcal{Y} \in Modules$, nous notons $\widetilde{\mathcal{Y}}$ la plus petite classe qui ait contenu $\mathcal{Y}$ telle que $\widetilde{\mathcal{Y}} \notin Modules$ (voir figure 8.5). Alors la partition courante est stable pour les ensembles pivots de la forme $N(y) \setminus \widetilde{\mathcal{Y}}$ où $y \in \widetilde{\mathcal{Y}}$. Soient $\mathcal{Y}$ et $\mathcal{Y}'$ deux classes de *Modules* telles que $\mathcal{Y} \neq \mathcal{Y}'$. Puisque $\widetilde{\mathcal{Y}}$ et $\widetilde{\mathcal{Y}'}$ ont été des classes de la partition, nous avons $\widetilde{\mathcal{Y}} \cap \widetilde{\mathcal{Y}'} = \emptyset$ ou $\widetilde{\mathcal{Y}} \subset \widetilde{\mathcal{Y}'}$. Puisque $\widetilde{\mathcal{X}}$ est la première classe ajoutée à *Modules*, il n'existe pas de classe $\mathcal{Y} \in Modules$ telle que $\widetilde{\mathcal{X}} \subset \widetilde{\mathcal{Y}}$:

- Supposons que $\widetilde{\mathcal{Y}} \cap \widetilde{\mathcal{X}} = \emptyset$. Soient $\mathcal{Z}$ une classe de la partition courante incluse dans $\widetilde{\mathcal{Y}}$ et z un sommet de $\mathcal{Z}$. Alors $\mathcal{X} \cap N(z) = \mathcal{X} \cap (N(z) \setminus \widetilde{\mathcal{Y}})$, donc $\mathcal{X}$ est stable pour l'ensemble pivot associé à z.

- Supposons que $\widetilde{\mathcal{Y}} \subset \widetilde{\mathcal{X}}$. Nous avons alors $\widetilde{\mathcal{Y}} \cap \mathcal{X} = \emptyset$. Soient $\mathcal{Z}$ une classe de la partition courante incluse dans $\widetilde{\mathcal{Y}}$ et z un sommet de $\mathcal{Z}$. Alors $\mathcal{X} \cap N(z) = \mathcal{X} \cap (N(z) \setminus \widetilde{\mathcal{Y}})$, donc $\mathcal{X}$ est stable pour l'ensemble pivot associé à z.

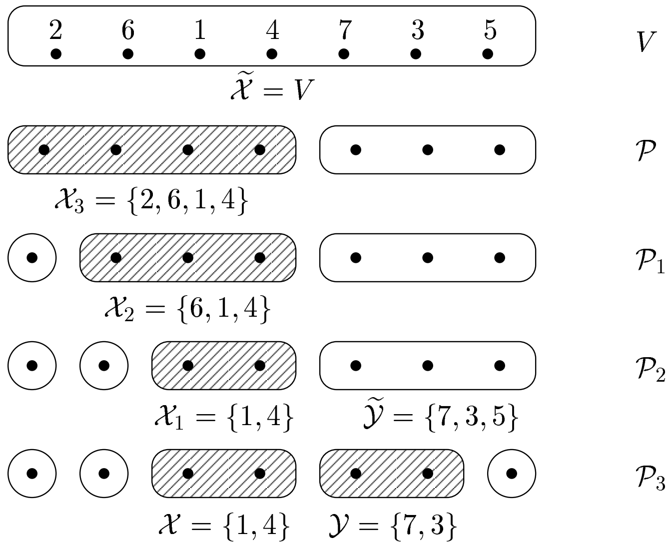


FIG. 8.5 – *Evolutions de la file Modules dans l'exemple de la figure 8.4 (les classes hachurées appartiennent à Modules).*

Donc $\mathcal{X}$ est stable pour les ensembles pivots associés à toutes les classes de la partition (celles de *Modules* et les autres). Donc $\mathcal{X}$ est un module. Nous pouvons alors choisir un seul sommet de $\mathcal{X}$ pour affiner $\mathcal{P}$ (*cf.* lemme 8.10). La même démonstration que ci-dessus montre alors qu'après affinage, selon une classe de *Modules*, toute partition modulaire est plus fine que $\mathcal{P}$.

Enfin, la structure de données implique que chaque étape d'affinage produit une partition compatible. La propriété A est préservée. Puisque initialement $\mathcal{P} = \mathcal{Q}$, la partition finale est compatible avec $\mathcal{Q}$.

Montrons que la partition finale vérifie la propriété B. Si une classe $\mathcal{X}$ n'appartient ni à *Pivots* ni à *Modules*, alors elle n'a pas été coupée depuis que la partition courante a été affinée selon les ensembles pivots associés à $\mathcal{X}$: $\mathcal{P}$ est stable pour $\mathcal{X}$. Donc si *Pivots* et *Modules* sont vides, $\mathcal{P}$ est stable pour toute classe $\mathcal{X}$. Autrement dit, toute classe est un module et $\mathcal{P}$ est une partition modulaire.

De plus, la propriété A montre que $\mathcal{P}$ est plus grossière que toutes les partitions modulaires pour $\mathcal{Q}$. Puisque c'est elle-même une partition modulaire, c'est l'unique partition modulaire maximum (*i.e.* avec la moins de classes). Le lemme 8.9 est donc prouvé. Puisque A montre aussi que $\mathcal{P}$ est compatible avec $\mathcal{Q}$, alors $\mathcal{P}$ est une partition fonctionnelle maximum pour la fonction f compatible avec $\mathcal{Q}$. □

Analyse de complexité : Lors du premier appel à la procédure *InitPartition*, l'initialisation de *Modules* et *Pivots* peut se faire en temps linéaire. Les appels suivants sont en temps constant. Le nombre d'appel à cette procédure est inférieur au nombre d'appel de procédures *AjoutPivot* et *EnsemblePivot*. Nous avons vu que la structure de données gérant l'ensemble $Modules$ permet de faire les opérations de retrait et d'échange d'un élément en temps constant. La procédure *InsertàDroite* est en temps constant.

Le calcul d'un ensemble pivot S et l'affinage selon S se font en $O(|S|)$. A chaque fois que la procédure *InitPartition* est appelée, la première classe $\mathcal{X}$ ajoutée à $module$ est choisie (c'est un module), et un ensemble $\{x\}$, où $x \in \mathcal{X}$, est ajouté à $Pivots$. Chaque appel à la procédure *InitPartition* se fait aussi en temps constant. Puisque $\mathcal{X}$ est un module, elle n'est plus jamais affectée par des opérations ultérieures. Il y a donc de l'ordre de $O(n)$ appels à *InitPartition*. Intéressons-nous au nombre de fois où un sommet peut être utilisé comme pivot.

Lorsqu'une classe $\mathcal{X}$ de $Pivots$ a été utilisée pour affiner la partition, toutes les classes de $\mathcal{P}$ sont stables pour $N(x)$ où $x \in X$. Ce n'est donc pas la peine d'utiliser à nouveau x comme pivot tant que la classe $\mathcal{X}$ n'a pas été elle même coupée. La règle d'ajout à $Pivots$ de la procédure *AjoutPivot* assure que la classe d'un sommet x ne sera pas rajoutée tant qu'elle n'aura pas une taille inférieure à la moitié de la classe qui la contenait la dernière fois que $N(x)$ a été utilisé pour affiner la partition. Cela implique que chaque voisinage soit utilisé au plus $\log n$ fois. Nous aboutissons donc à une complexité en $O(n + m \log n)$. $\square$

Théorème 8.11 *Etant donnée une partition $\mathcal{Q}$ des sommets d'un graphe G, l'algorithme 22 calcule une partition modulaire maximum pour $\mathcal{Q}$ en $O(n + m \log n)$.*

Evoquons brièvement comment l'algorithme 22 pourrait être utilisé pour calculer la décomposition par substitution d'un graphe. Pour être plus précis, l'idée que nous développons permet de calculer une permutation factorisante d'un graphe. En utilisant l'algorithme de Capelle [Cap96], nous pouvons donc reconstruire l'arbre de décomposition.

Le problème est de proposer une partition initiale. Pour calculer la partition en modules maximaux de G, il suffit de fournir à l'algorithme 22 une partition de la forme : $V \setminus M, M$ où M est un module maximal non-trivial.

Une fois que le processus s'arrête, nous pouvons alors le relancer pour chaque sous-graphe $G[\mathcal{X}_i]$ où $\mathcal{X}_i$ est une classe de la partition $\mathcal{P}$ obtenue. Le problème réside donc dans le calcul d'un module maximal non-trivial pour tous les sous-graphes de G induits par un module. Pour obtenir une complexité raisonnable, il faut pouvoir le faire en $O(|M|)$. Ce point n'est pas simple à résoudre.

8.2 Reconnaissance des cographes

Nous proposons dans cette section un nouvel algorithme de reconnaissance des cographes en $O(n+m\log n)$. Corneil, Pearl et Stewart en avait proposé un linéaire [CPS85]. Malgré une moins bonne complexité, l'algorithme que nous proposons nous semble intéressant pour sa simplicité. En effet, il s'agit encore une fois d'un processus d'affinage de partition.

Pour cette application, nous ne connaissons pas systématiquement une manière efficace de choisir un pivot qui affine strictement la partition courante. Nous devons donc, pour obtenir la complexité annoncée, à nouveau utiliser la règle de Hopcroft. Nous verrons cependant comment "une fois sur deux", il est possible de choisir un pivot à l'aide d'un choix arbitraire d'un sommet dans une classe. Cet algorithme constitue donc une étape intermédiaire vers une complexité linéaire.

8.2.1 Rappel sur les cographes

Pour définir les cographes, nous avons besoin des deux opérations suivantes :

1. *Composition parallèle :* c'est l'union disjointe de deux graphes G_1 et G_2.
2. *Composition série :* c'est le produit de deux graphes G_1 et G_2 (*i.e.* tout sommet x_1 de G_1 est connecté à tout sommet x_2 de G_2).

Définition 8.12 *La classe des* ***cographes*** *est la plus petite classe de graphes contenant le graphe réduit à un sommet et, fermée pour les opérations de composition série et parallèle.*

Les cographes ont aussi été caractérisés de la manière suivante :

Théorème 8.13 *[Sei74] Un* cographe *ne contient pas de sous-graphe isomorphe au P_4.*

La propriété suivante due à Sumner, nous renseigne sur la structure de l'arbre de décomposition modulaire associé à un cographe.

Lemme 8.14 *[Sum73] Un graphe premier contient un sous-graphe isomorphe à un P_4.*

Nous en déduisons que les cographes admettent un arbre de décomposition n'ayant aucun noeud premier. Seules les décompositions séries et parallèles sont utilisées. Un tel arbre est appelé *coarbre* (voir figure 8.6). Il faut noter que les noeuds internes d'une branche d'un coarbre allant d'une feuille à la racine sont alternativement étiquetés parallèle et série.

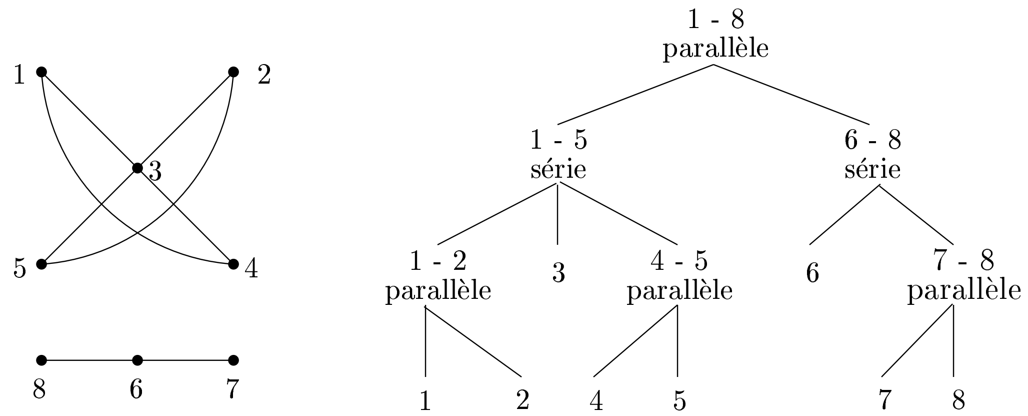


FIG. 8.6 – *Un cographe et le coarbre associé*

Le coarbre permet très facilement de tester l'adjacence entre deux sommets x et y : x et y sont adjacents si et seulement si leur plus petit ancêtre commun est un noeud série.

8.2.2 L'algorithme

L'algorithme que nous proposons est basé sur un paradigme intéressant. Il a déjà été utilisé par d'autres auteurs, en particulier pour la reconnaissance des graphes triangulés [RTL76, TY84] (voir chapitre 1). Il permet de concevoir des algorithmes très simples. Il fonctionne en deux étapes :

1. calcul d'une propriété caractérisant la classe de graphes étudiée en supposant que le graphe donné appartienne à cette classe.

2. vérification du résultat.

Durant le calcul, nous supposons que tout se passe bien. Nous ne faisons aucun test. Ils sont tous effectués durant l'étape de vérification. Nous utiliserons ce paradigme à nouveau dans la suite de cette partie. Il nous servira pour l'orientation transitive d'un graphe de comparabilité et la reconnaissance des graphes d'intervalles. L'avantage de ce principe est de bien dissocier les deux problèmes, calcul d'une propriété caractéristique et vérification. Nous pouvons ainsi mieux cerner où se trouvent les difficultés. Par exemple, nous verrons que, pour l'orientation transitive, la première étape peut se faire en linéaire mais pas la seconde.

L'algorithme présenté calcule une permutation factorisante si et seulement si le graphe en entrée est un cographe. Pour la phase de vérification, nous pouvons construire un coarbre. Il est alors facile de tester si le coarbre obtenu correspond au graphe initial. Pour obtenir le coarbre, deux solutions sont envisageables. Soit nous utilisons l'algorithme proposé par Capelle [Cap96] qui calcule l'arbre de décomposition (donc le coarbre) en $O(n+m)$. Il est alors facile de tester si le coarbre obtenu correspond au graphe initial. Soit nous modifions légèrement l'algorithme que nous allons présenter pour qu'il construise un coarbre si et seulement si le graphe donné est un cographe. La vérification se fait alors facilement.

L'inconvénient d'une telle méthode par rapport à l'algorithme proposé par Corneil, Pearl et Stewart [CPS85] est qu'elle n'admet par l'incrémentalité (*i.e.* si nous rajoutons ou supprimons un sommet au graphe, nous devons repartir à zéro). En revanche, elle mène à un algorithme très simple à comprendre et donc à implémenter.

Avant de présenter l'algorithme en détails, donnons en l'idée principale. Etant donné un sommet quelconque x d'un cographe $G = (V, E)$, il est possible de permuter les feuilles du coarbre de telle manière que tous les voisins de x soient à droite de x et tous ses non-voisins à sa gauche (voir figure 8.7).

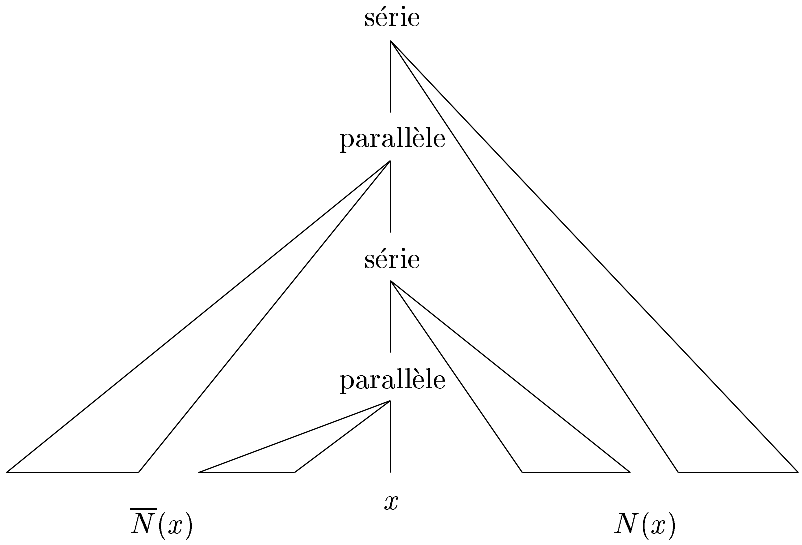


FIG. 8.7 – *Réarrangement d'un coarbre*

Nous savons aussi que sur la branche allant de x à la racine du coarbre, les noeuds séries et parallèles alternent. Donc les sous-arbres issus d'un noeud série (*resp.* parallèle) sur la branche de x à la racine sont placés à gauche (*resp.* droite).

Le problème consiste alors à regrouper consécutivement les feuilles de chacun de ces sous-arbres. Ceci peut être fait en affinant la classe des non-voisins de x (*resp.* des voisins) avec les voisinages des voisins de x (*resp.* non-voisins).

Une fois ce travail effectué, il ne reste plus qu'à relancer le processus dans chacune des classes (*i.e.* des sous-arbres issus d'un noeud entre x et la racine du coarbre) calculées.

L'algorithme que nous présentons utilise la règle de Hopcroft pour lancer le processus lorsque $\overline{N}(x), x, N(x)$ ont été séparés. Soient $\mathcal{Z}$ la plus petite classe parmi $N(x)$ et $\overline{N}(x)$ et $\mathcal{Z}'$ la plus grande. La classe $\mathcal{Z}$ est ajoutée à $Pivots$ et $\mathcal{Z}'$ à $Modules$. En revanche, quand $\mathcal{Z}'$ a entièrement été affinée par les sommets de $\mathcal{Z}$, il suffit de choisir un sommet dans chacune des sous-classes de $\mathcal{Z}'$ (qui appartiennent toutes à $Modules$) pour affiner $\mathcal{Z}$. Nous sommes donc dans une situation où, pour sélectionner les pivots, nous utilisons alternativement la règle de Hopcroft et le choix arbitraire d'un sommet dans une classe.

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans $Pivots$ ou $Modules$ qui affine strictement $\mathcal{P}$.*

- ***A:*** *Une permutation factorisante est compatible avec $\mathcal{P}$ si G est un cographe.*
- ***B:*** *Toutes les classes sont des singletons.*

Preuve : Supposons que le graphe G soit un cographe. Nous allons montrer que les deux premiers appels à la procédure *InitPartition* calculent la partition modulaire maximum $\mathcal{P}$ compatible avec la partition $\{\overline{N}(x), \{x\}, N(x)\}$ où x est le sommet choisi par le premier appel à *InitPartition*. Supposons qu'une permutation factorisante soit compatible avec $\mathcal{P}$. Puisque les cographes sont des graphes héréditaires. En calculant une permutation factorisante de chaque sous-graphe $G[\mathcal{X}]$ où $\mathcal{X}$ est une classe de la partition courante, nous obtenons une permutation factorisante de G. Il suffit alors de montrer que les opérations effectuées avant le troisième appel à *InitPartition* soient correctes.

Nous montrons maintenant par induction que la propriété A est vérifiée après chaque étape d'affinage de l'algorithme. Tout d'abord puisqu'une permutation factorisante peut être obtenue en lisant les feuilles du coarbre de gauche à droite, il existe une permutation factorisante compatible avec la partition $\overline{N}(x), \{x\}, N(x)$ (voir figure 8.7). Donc l'affinage par le premier appel à la procédure *InitPartition* préserve la propriété A. Montrons qu'elle l'est avant une étape d'affinage par un élément $\mathcal{C}_p$ de $Pivots$. Nous avons deux cas :

Algorithme 23: Calcul d'une permutation factorisante d'un cographe

Données : Un graphe $G = (V, E)$

Résultat : Une permutation factorisante $\mathcal{P}$ si et seulement si G est un cographe

début

  $\mathcal{P} \leftarrow V$;

  exécuter l'algorithme 9 avec la procédure 10 les définitions suivantes (les éléments de $Pivots$ sont des sommets de G);

fin

Procedure *InitPartition* $(\mathcal{P})$

  au $2i+1$*-ème appel:*

    si $\mathcal{P}$ *ne contient que des singletons* **alors**

      fin du processus, **retourner** $\mathcal{P}$;

    pour chaque *classe* $\mathcal{X} \in \mathcal{P}$ *telle que* $|\mathcal{X}| > 1$ **faire**

      si $premierPivot(\mathcal{X})$ *n'existe pas* **alors**

        choisir un sommet quelconque $x \in \mathcal{X}$;

      sinon $x \leftarrow premierPivot(\mathcal{X})$;

      remplacer $\mathcal{X}$ dans $\mathcal{P}$ par $\mathcal{X} \cap \overline{N}(x), \{x\}, \mathcal{X} \cap N(x)$;

      soit $\mathcal{Z}$ la plus petite classe parmi $\mathcal{X} \cap \overline{N}(x)$ et $\mathcal{X} \cap N(x)$ et $\mathcal{Z}'$ la plus grande;

      ajouter $\mathcal{Z}$ à $Pivots$;

      ajouter $\mathcal{Z}'$ à $Modules$;

  au $2i$*-ème appel:*

    tant que $Modules \neq \emptyset$ **faire**

      choisir une classe $\mathcal{X} \in Modules$ et un sommet quelconque $x \in \mathcal{X}$;

      ajouter $\{x\}$ à $Pivots$;

      $premierPivots(\mathcal{X}) \leftarrow \{x\}$;

Procedure *EnsemblePivot* $(p = (x, \mathcal{X}))$

  retourner $N(x) \setminus \mathcal{X}$;

Procedure *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$

  retourner vrai;

Procedure *AjoutPivot* $(\mathcal{X}, \mathcal{X}_a)$

  si $\mathcal{X} \in Modules$ **alors**

    ajouter $\mathcal{X}_a$ à $Modules$;

1. *Soit $\mathcal{C}_p$ une classe de $\mathcal{P}$ (i.e. la classe $\mathcal{Z}$ ajoutée par le premier appel à InitPartition) :* Pour fixer les idées, supposons sans perte de généralités que $\mathcal{Z} = \overline{N}(x)$. Soit $z \in \mathcal{Z}$. Le plus petit ancêtre commun N à x et z dans le coarbre est un noeud parallèle. L'ensemble $N(z) \cap N(x)$ correspond aux feuilles des sous-arbres d'un noeud série entre N et la racine du coarbre. Tout sommet y, appartenant à un sous-arbre issu d'un noeud série entre x et N, a, comme plus petit ancêtre commun avec z, le noeud N. Cela signifie que z et y ne sont pas adjacents. Pour qu'une permutation factorisante soit compatible avec la partition affinée par $N(z)$, il faut donc mettre à droite le voisinage de z et à gauche son non-voisinage. C'est ce que fait l'algorithme 23. La propriété A est préservée.

2. *Supposons que $\mathcal{C}_p$ soit un singleton (i.e. un sommet ajouté à Pivots par le second appel à InitPartition) :* A l'issue de l'affinage de $\mathcal{P}$ selon les voisinages de tous les sommets de $\mathcal{Z} = \overline{N}(x)$, nous obtenons les modules maximaux inclus dans $N(x)$. D'une part, chaque classe incluse dans $N(x)$ est stable pour le voisinage des sommets de $\overline{N}(x)$. Mais si deux sommets y et y' appartiennent à deux classes différentes, alors leur plus petit ancêtre commun dans le coarbre est un noeud série (voir figure 8.7). Donc il existe toutes les arêtes possibles entre les classes incluses dans $N(x)$.

   Si nous faisions le même traitement avec les sommets de $\mathcal{Z}'$, nous obtiendrions aussi les modules maximaux inclus dans $N(x)$. La même preuve suffirait. Ceci dit, nous ferions du travail inutile. Soient y et y' deux sommets appartenant à l'une de ces classes $\mathcal{Y}$. Alors $N(y) \cap N(x) = N(y') \cap N(x)$ ($\mathcal{Y}$ est un module). Il suffit donc d'affiner la partition avec un seul sommet par classe. Tous les modules de $\mathcal{Z} = \overline{N}(x)$ sont contenus dans $Modules$. Le second appel à la procédure *InitPartition* sélectionne un sommet par module de $\overline{N}(x)$ et l'ajoute à $Pivots$ pour relancer le processus. L'étape d'affinage est ensuite la même que précédemment. La propriété A est donc bien préservée.

Le processus s'arrête lorsque toutes les classes sont des singletons. Puisqu'à chaque étape la propriété A a été préservée, nous obtenons une permutation factorisante de G. □

Analyse de complexité : Les procédures *AjoutPivot* et *InsertàDroite* se font en temps constant. Comme dans tous les algorithmes vus jusqu'à présent, l'affinage de la partition par un ensemble S se fait en $O(|S|)$. Ici les ensembles pivots sont les voisinages des sommets. La procédure *EnsemblePivot* peut aussi se faire en $O(|N(x)|)$ où x est le sommet pivot. Il faut donc estimer le nombre de fois qu'un sommet x peut être ajouté à $Pivots$.

Si un sommet x est utilisé par la procédure *InitPartition* pour affiner une classe $\mathcal{X}$ en $\overline{N}(x) \cap \mathcal{X}, \{x\}, N(x) \cap \mathcal{X}$, alors il ne le sera plus par la suite. De même si un sommet x est défini comme le *premierPivot* d'une classe $\mathcal{X}$, alors il ne sera utilisé plus que 2 fois : une fois pour affiner les classes extérieures et une seconde fois pour affiner sa propre classe. Avant d'être utilisé d'une de ces deux manières, un sommet x peut être rajouté à *Pivots* par la procédure *InitPartition* s'il appartient à une sous-classe de taille inférieure à la moitié de la classe affinée (règle de Hopcroft). Cela assure que tout sommet est ainsi ajouté à *Pivots* au plus $\log n$ fois. Nous pouvons donc utiliser un sommet au plus $2 + \log n$ fois comme pivot.

Puisque les voisinages des sommets sont de l'ordre de $O(m)$, nous obtenons finalement une complexité en $O(n + m \log n)$ pour l'algorithme 23. □

Théorème 8.15 *Soit un graphe G. L'algorithme 23 calcule une permutation factorisante de G si et seulement si G est un cographe. Sa complexité est $O(n + m \log n)$.*

8.3 Orientation transitive

Un graphe orienté $G = (V, E)$ est *transitif* si, pour tout triplet de sommets a, b, c, l'existence des arcs (a, b) et (b, c) implique l'existence de l'arc (a, c). Un sommet n'ayant que des arcs entrants (*resp.* sortants) est appelé un *puits* (*resp. source*). Etant donné une *orientation transitive* d'un graphe, une autre orientation transitive peut être obtenue en renversant le sens de tous les arcs. Nous parlerons d'*orientation inverse*.

Le problème qui nous intéresse est le suivant : *étant donnée un graphe non-orienté $G = (V, E)$, existe-t-il une orientation transitive des arêtes de E ?* Notons que ce problème est équivalent à la recherche d'une orientation transitive sans circuit.

Définition 8.16 *Un graphe non-orienté $G = (V, E)$ est un graphe de* ***comparabilité*** *si et seulement si il existe une orientation transitive de l'ensemble E de ses arêtes.*

Le problème énoncé ci-dessus est donc équivalent à reconnaître les graphes de comparabilité. Cette classe de graphe est importante et a longuement été étudiée [Gal67, Gol80]. Elle contient en particulier les graphes de permutation et les graphes d'intervalles. Nous étudierons cette dernière classe dans la seconde section de ce chapitre.

Bien que Mc Connell et Spinrad [MS94, MS97] aient montré que l'orientation transitive d'un graphe peut se faire en temps linéaire, leur algorithme est très compliqué. En fait, ils ont montré comment utiliser un algorithme linéaire de décomposition modulaire pour obtenir un choix “intelligent” des pivots. Face à cette difficulté, le choix d'un algorithme raisonnable à implémenter pour ce problème reste sans doute la version utilisant la règle de Hopcroft. Néanmoins, jusqu'alors cette version utilisait toujours la décomposition modulaire en pré-traitement. Nous montrons, dans cette section, comment éviter cette étape grâce à Lex-BFS. Nous présentons brièvement le principe de l'algorithme de Mc Connell et Spinrad qui calcule l'orientation transitive d'un graphe de co-comparabilité (*i.e.* complémentaire d'un graphe de comparabilité). Puis nous étudions certaines propriétés des parcours Lex-BFS. Nous pourrons alors détailler l'algorithme complet d'orientation transitive.

8.3.1 Rappels sur l'orientation transitive

Soit $G = (V, E)$ un graphe non-orienté, si $(a, b) \in E$, $(b, c) \in E$ et $(a, c) \notin E$, alors nous dirons que (a, b) *force* (b, c) : *i.e.* si l'arête (a, b) est orientée de a vers b (*resp.* b vers a), alors l'arête (b, c) est orientée de c vers b (*resp.* b vers c) dans toute orientation transitive. Un ensemble d'arêtes se forçant deux à deux est appelé *classe de forçage*. Les classes de forçage ont été étudiées par Gallai [Gal67] et sont à la base de tout travail sur l'orientation transitive d'un graphe. En fait, il est montré que les arêtes de deux classes de forçage différentes peuvent être orientées de manière indépendante.

Le lemme suivant est un résultat connu, il joue un rôle important dans les algorithmes d'orientation transitive :

Lemme 8.17 *Soit M un module d'un graphe de comparabilité G, alors les arêtes du sous-graphe $G[M]$ peuvent être orientées indépendamment des autres arêtes de G.*

Preuve : Il suffit de montrer que les arêtes internes au graphe $G[M]$ ne sont pas dans les mêmes classes de forçage que les autres arêtes de G. Soient a et b deux sommets de M adjacents. Soit e une arête de G n'appartenant pas à $G[M]$. Si ni a, ni b n'est une extrémité de e, alors par définition e ne force pas (a, b). Supposons que a soit une extrémité de e. Soit c l'autre sommet de e, alors $c \notin M$. Puisque c est voisin de $a \in M$, tous les sommets de M sont adjacents à c, et en particulier le sommet b. Donc a, b, c induisent un sous-graphe complet et l'arête (a, b) n'est pas forcée par $e = (a, c)$. □

Supposons que nous disposions d'une partition en modules $\mathcal{Q} = \{M_1, \ldots, M_k\}$ de l'ensemble des sommets d'un graphe de comparabilité $G = (V, E)$. Soit $G/\mathcal{P}$ le graphe obtenu à partir de G en contractant chacun des modules de $\mathcal{Q}$ en un seul sommet. Alors calculer une orientation transitive de G peut se faire en calculant d'une part une orientation transitive de $G/\mathcal{Q}$ et d'autre part une orientation transitive de chaque sous-graphe $G[M_i]$.

Mc Connell et Spinrad ont ainsi réduit le problème de l'orientation d'un graphe de co-comparabilité au cas des graphes premiers. Expliquons leur démarche.

Soit G un graphe de co-comparabilité premier. Ils calculent une extension linéaire d'une orientation transitive de $\overline{G}$, c'est-à-dire un ordre total sur les sommets tel que les non-arêtes de G (*i.e.* les arêtes de $\overline{G}$) sont orientées de la droite vers la gauche. Pour cela, ils utilisent une partition ordonnée.

Le processus est lancé avec la partition $\mathcal{P} = \{V \setminus \{v\}, \{v\}\}$ où v est une source de $\overline{G}$. Nous expliquerons dans la prochaine section comment calculer x. A chaque étape, la partition est affinée par un sommet pivot x. Soit $\mathcal{X}$ la classe contenant x, l'ensemble pivot est $S = N(x) \setminus \mathcal{X}$. Chaque classe $\mathcal{Y}$ est donc partitionnée en $\mathcal{Y}_1 = \mathcal{Y} \cap S$ et $\mathcal{Y}_2 = \mathcal{Y} \setminus S$. Notons que $\mathcal{X}$ est stable pour S. Afin de respecter l'orientation des arêtes de $\overline{G}$, si $\mathcal{Y}$ est à droite de $\mathcal{X}$, alors $\mathcal{Y}_1$ est placée à droite de $\mathcal{Y}_2$, sinon $\mathcal{Y}_1$ est placée à gauche de $\mathcal{Y}_2$ (voir figure 8.8).

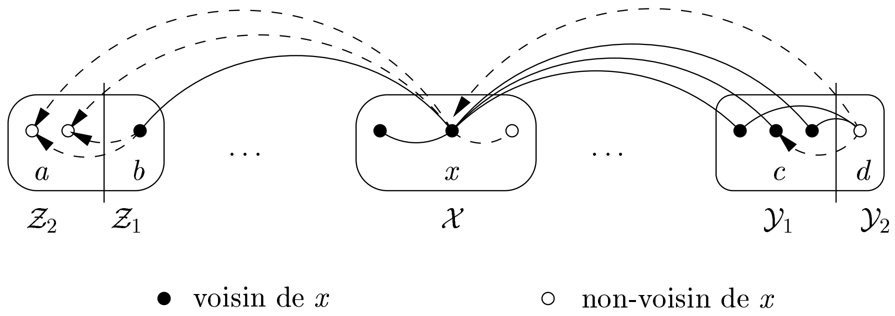


FIG. 8.8 – *Affinage des classes de la partition $\mathcal{P}$ par un pivot x. L'orientation des non-arêtes (a, b) et (c, d) est forcée par (x, a) et (x, d).*

La validité de leur algorithme peut être montrée par induction de la manière suivante : supposons qu'avant d'affiner $\mathcal{P}$ avec le sommet pivot x, les arêtes de $\overline{G}$ (*i.e.* les non-arêtes de G) n'ayant pas leurs extrémités dans une même classe de $\mathcal{P}$ puissent être orientées de la droite vers la gauche. Puisque l'algorithme commence avec la partition $\{V \setminus \{v\}, \{v\}\}$ où v est une source, le cas de base est vrai. Montrons que cette propriété est toujours vérifiée après l'affinage par x.

Soit $\mathcal{Y}$ une classe partitionnée en $\mathcal{Y}_a, \mathcal{Y}_b$ à cette étape. Nous devons nous intéresser aux non-arêtes de G ayant une extrémité dans $\mathcal{Y}_a$ et l'autre dans $\mathcal{Y}_b$. Soit e une telle non-arête. Si $\mathcal{X}$, la classe contenant x, est à gauche (*resp.* à droite) de $\mathcal{Y}$, alors les non-arêtes entre x et les sommets de $\mathcal{Y}_b$, orientées de gauche à droite, forcent l'orientation de e de la gauche vers la droite. Puisque G est premier, il existe toujours un sommet pivot qui permet d'affiner une classe non-singleton. Donc lorsque la partition ne contient que des singletons toutes les non-arêtes sont orientées de la gauche vers la droite.

8.3.2 Ordres Lex-BFS

Lex-BFS est un algorithme de partitionnement (voir chapitre 6). Il calcule un ordre total sur les sommets que nous appelons *ordre Lex-BFS*. Cet ordre est calculé à "l'envers" : *i.e.* le dernier sommet visité est le sommet numéroté 1 par Lex-BFS. Soit $\mathcal{P} = \{\mathcal{X}_1, \dots, \mathcal{X}_k\}$ la partition courante d'une exécution de Lex-BFS. Alors nous avons la propriétés suivante :

Lemme 8.18 *Si toutes les classes de $\mathcal{P}$ sauf $\mathcal{X}_1$ sont des singletons, alors $\mathcal{X}_1$ est un module.*

Preuve : Puisque les classes $\mathcal{X}_2, \dots, \mathcal{X}_k$ sont des singletons, cela signifie que la partition $\mathcal{P}$ est stable pour le voisinage de n'importe quel sommet de $S = \mathcal{X}_2 \cup \ldots \cup \mathcal{X}_k$. En particulier, $\mathcal{X}_1$ est stable pour tout $N(x)$ où $x \in S$. Autrement dit les sommets de $\mathcal{X}_1$ ont tous le même voisinage à l'extérieur de $\mathcal{X}_1$. Donc $\mathcal{X}_1$ est un module. □

Il est facile de détecter cette situation : elle se produit lorsque le sommet pivot est choisi dans la première classe de la partition $\mathcal{P}$.

Nous l'avons vu, l'algorithme d'orientation transitive proposé par Mc Connell et Spinrad commence dans un premier temps par chercher une source du graphe. Le lemme 8.19 montre que Lex-BFS peut remplir ce rôle :

Lemme 8.19 *Si G est un graphe de co-comparabilité premier et si $\sigma = [x_1, \dots, x_n]$ est un ordre Lex-BFS des sommets de G, alors il existe une orientation transitive de $\overline{G}$ où x_1 est une source.*

Preuve : Soit $S = \mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_k$ la suite des classes contenant x_1 pendant l'exécution de Lex-BFS où $\mathcal{X}_1 = V$ et $\mathcal{X}_k = \{x_1\}$. Puisque x_1 est un élément de ces classes, elles sont toujours en première position dans la partition courante de Lex-BFS. A chaque fois, pour $1 \leqslant i \leqslant k+1$, $\mathcal{X}_i$ est affinée en $\mathcal{X}_{i+1}$ et une autre classe $\mathcal{Y}$ par un sommet pivot z n'appartenant pas à $\mathcal{X}_i$ (sinon le graphe ne serait pas premier, *cf.* lemme 8.18). Donc les sommets de $\mathcal{Y}$ sont adjacents à z et ceux de $\mathcal{X}_{i+1}$ ne le sont pas.

Supposons par hypothèse d'induction, qu'avant l'affinage de $\mathcal{X}_i$ par $N(z)$, il existe une orientation transitive qui oriente toutes les non-arêtes de G appartenant à $\{x_1\}\times(V\setminus\mathcal{X}_i)$ vers x_1. Alors toute non-arête (y, x_1) de G, où $y \in \mathcal{Y}$, est forcée par la non-arête (z, x_1) (car (z, y) est une arête de G). Donc (y, x_1) doit être orientée vers x_1. Ceci montre que l'hypothèse d'induction est aussi vraie pour $\mathcal{X}_{i+1}$.

Le cas de base est donné par la partition $\mathcal{P} = \{\mathcal{X}_2 = V \setminus \{x_n\}, \{x_n\}\}$. Si l'arête (x_1, x_n) existe, elle peut être orientée dans n'importe quelle direction puisque l'inverse d'une orientation transitive est aussi une orientation transitive. $\square$

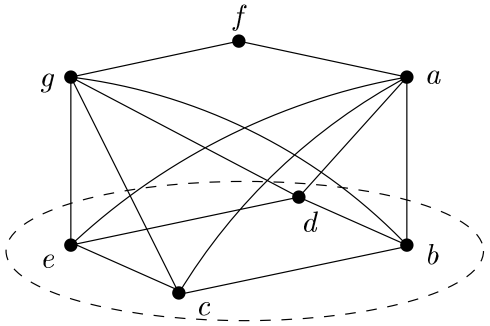
Nous souhaitons éviter la décomposition modulaire pour ne pas se restreindre aux graphes premiers. Il faut donc généraliser le lemme 8.19. Soit $\mathcal{Q}$ une partition en modules de G. Il est impensable d'exécuter Lex-BFS sur chacun des modules de $\mathcal{Q}$, la complexité serait de l'ordre de $O(nm)$. Soit σ un ordre Lex-BFS. Pour chaque module $M \in \mathcal{Q}$, le *rang d'apparition* de M dans σ est $max\{\Sigma^{-1}(x) : x \in M\}$. Le lemme 8.20 permet de résoudre le problème évoqué ci-dessus.

Lemme 8.20 *Soit G un graphe non-orienté. Alors un ordre Lex-BFS σ vérifie les deux propriétés suivantes :*

1. *Si M est un module de G, alors σ induit un ordre Lex-BFS sur le sous-graphe $G[M]$.*
2. *Si $\mathcal{Q}$ est une partition modulaire des sommets de G, ordonner les modules de $\mathcal{Q}$ par leur rang d'apparition dans σ produit un ordre Lex-BFS de $G/\mathcal{Q}$.*

Preuve : En ce qui concerne le premier point, il faut simplement noter que tout pivot $z \notin M$ ne changera pas l'ordre relatif entre les éléments de M. En effet, z est soit adjacent à tous les sommets de M soit à aucun. Donc l'ordre entre les sommets de M ne provient que des opérations effectuées avec des sommets de M. C'est donc un ordre Lex-BFS du sous-graphe $G[M]$.

Soit $\mathcal{Q} = \{\mathcal{Y}_1, \ldots, \mathcal{Y}_k\}$ la partition modulaire. Soit y_i le premier sommet visité de $\mathcal{Y}_i$ par σ (*i.e.* le premier sommet de $\mathcal{Y}_i$ ayant servi à pivoter ou encore le sommet de plus grand numéro). Nous analysons l'ordre relatif aux sommets $\{y_1, \ldots, y_k\}$. Puisque tous les sommets d'un module ont le même voisinage à l'extérieur de M, tous les sommets de $\mathcal{Y}_i$ auront le même effet sur l'ordre relatif des sommets des autres modules. Il est donc suffisant de travailler sur le graphe $G/\mathcal{Q}$. Donc σ induit un ordre Lex-BFS sur $G/\mathcal{Q}$. $\square$



Ordre Lex-BFS sur G :

$\sigma = [g, f, e, d, c, b, a]$

$M = \{e, d, c, b\}$ est un module

Ordre Lex-BFS sur $G[M]$:

$\sigma_M = [e, d, c, b]$

FIG. 8.9 – *Un ordre Lex-BFS est héréditaire sur les modules du graphe.*

Généralisons le lemme 8.18 aux graphes non-premiers. Ce résultat est la base de la généralisation de l'algorithme de Mc Connell et Spinrad que nous proposons.

Théorème 8.21 *Si G est un graphe de co-comparabilité et si x_1 est le premier sommet d'un ordre Lex-BFS σ, alors il existe une orientation transitive de $\overline{G}$ où x_1 est une source.*

Preuve : Le théorème est déjà démontré par le lemme 8.18 dans le cas où G est un graphe premier. Supposons donc le contraire. Nous allons faire une preuve par induction. Faisons l'hypothèse que la propriété soit vraie pour un graphe ayant moins de sommet que G. Le cas de base est donné en considérant le plus petit sous-graphe premier de G contenant x_1.

Soit $\mathcal{X} \neq V$ le module maximal contenant le sommet x. Soit $\mathcal{Y}$ le module de cardinal maximum inclus dans $V \setminus \mathcal{X}$. Puisque G n'est pas un graphe premier, au moins un des modules parmi $\mathcal{X}$ et $\mathcal{Y}$ n'est pas un singleton. Considérons la partition en modules de V, $\mathcal{Q} = \{\mathcal{X}, \mathcal{Y}, \{y_1\}, \ldots, \{y_p\}\}$ où $\{y_1, \ldots, y_p\}$ sont les sommets de $V \setminus (\mathcal{X} \cup \mathcal{Y})$. Clairement les graphes $G/\mathcal{Q}$ et $G[\mathcal{X}]$ ont moins de sommets que G.

Puisque x_1 est aussi le premier sommet du sous-ordre induit par σ sur chacun de ces graphes, l'hypothèse d'induction montre que x_1 peut être choisi comme une source dans une orientation des sous-graphes respectifs. Le lemme 8.20 montre que ces deux orientations sont indépendantes (*i.e.* les classes de forçage sont disjointes). En les réunissant sur G, nous obtenons donc une orientation transitive où x_1 est une source. □

8.3.3 L'algorithme d'orientation transitive

Nous présentons maintenant en détails l'algorithme d'orientation transitive du complémentaire d'un graphe de co-comparabilité. Il faut le comparer avec l'algorithme 22. Les règles de pivots sont exactement les mêmes. Nous nous baserons sur le lemme 8.10 pour relancer le processus.

Algorithme 24: Orientation transitive

Données : Un graphe $G = (V, E)$

Résultat : Une extension linéaire $\mathcal{P}$ de $\overline{G}$ si et seulement si G est un graphe de co-comparabilité

début

$\mathcal{P} \leftarrow \{V\}$ et $Modules \leftarrow V$;
soit $\sigma = [x_1, \ldots, x_n]$ un ordre Lex-BFS sur G;
exécuter l'algorithme 9 avec la procédure 10 et les définitions suivantes (les éléments de $Pivots$ sont des sommets de G);

fin

Procedure *InitPartition* ($\mathcal{P}$)

si *toutes les classes sont des singletons* **alors**
fin du processus et **retourner** $\mathcal{P}$;
sinon
choisir une classe $\mathcal{X} \in Modules$;
soit x le plus petit sommet de $\mathcal{X}$ dans σ;
remplacer dans $\mathcal{P}$ la classe $\mathcal{X}$ par $\mathcal{X} \setminus \{x\}, \{x\}$;
ajouter $\{x\}$ à $Pivots$;

Procedure *EnsemblePivot* ($p = (x, \mathcal{X})$)

retourner $N(x) \setminus \mathcal{X}$;

Procedure *InsertàDroite* ($\mathcal{X}, \mathcal{X}_a, p$)

soit x un élément de $\mathcal{X}$ **retourner** (x est à gauche de p);

Procedure *AjoutPivot* ($\mathcal{X}, \mathcal{X}_a$)

soit $\mathcal{Z}$ (*resp.* $\mathcal{Z}'$) la plus petite (*resp.* grande) classe parmi $\mathcal{X}$ et $\mathcal{X}_a$;
si $\mathcal{X} \in Pivots$ **alors** ajouter $\mathcal{X}_a$ à $Pivots$;
sinon
si $\mathcal{X} \in Modules$ **alors** échanger $\mathcal{X}$ par $\mathcal{Z}'$ dans $Modules$;
sinon ajouter $\mathcal{Z}'$ à $Modules$;
ajouter $\mathcal{Z}$ à $Pivots$;

Nous avions évoqué pour l'algorithme 22 la difficulté de l'étendre au calcul complet de la décomposition par substitution. Nous ne savions pas comment calculer récursivement une partition initiale pour relancer le processus dans chaque module. Dans le cas de l'orientation transitive, nous pouvons utiliser les propriétés de Lex-BFS que nous venons de voir.

L'algorithme 24 utilise à nouveau le paradigme évoqué à la section 8.2. Il calcule une extension linéaire de $\overline{G}$ si et seulement si G est un graphe de co-comparabilité. Il reste alors à tester le résultat. Actuellement, nous ne savons pas résoudre ce problème avec une complexité meilleure que celle du produit de matrice.

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots ou Modules qui affine strictement $\mathcal{P}$.*

- ***A:*** *Il existe une orientation transitive de $\overline{G}$ où les non-arêtes de G reliant des sommets de deux classes sont orientées de droite à gauche.*
- ***B:*** *Toutes les classes sont des singletons.*

Preuve : Montrons que chaque appel à la procédure *InitPartition* préserve la propriété A. Le théorème 8.21 montre que c'est vrai pour le premier appel. Pour les autres, nous choisissons à chaque fois la classe $\mathcal{X} \in Modules$. Le lemme 8.10 garanti que $\mathcal{X}$ est un module. Donc le lemme 8.20 montre alors la validité de la procédure *InitPartition* vis-à-vis de la propriété A.

Il reste à montrer que chaque étape d'affinage préserve aussi la propriété A. Nous reprenons ici le raisonnement tenu à la fin de la section 8.3.1. Supposons qu'avant d'affiner $\mathcal{P}$ par un ensemble S, les non-arêtes de G peuvent être orientées de la droite vers la gauche. Le cas de base est montré par la validité de la procédure *InitPartition* qui isole toujours une source (du graphe ou du module). Montrons donc que cette propriété est aussi vrai après l'affinage de $\mathcal{P}$ par $S = N(x) \setminus \mathcal{X}$, où $\mathcal{X}$ est la classe de $\mathcal{P}$ contenant x.

Soit $\mathcal{Y}$ une classe partitionnée en $\mathcal{Y}_a = \mathcal{Y} \cap S$ et $\mathcal{Y}_b = \mathcal{Y} \setminus S$ à cette étape. Nous devons nous intéresser aux non-arêtes de G ayant une extrémité dans $\mathcal{Y}_a$ et l'autre dans $\mathcal{Y}_b$. Soit e une telle non-arête. Si $\mathcal{X}$, la classe contenant x, est à gauche (*resp.* à droite) de $\mathcal{Y}$, alors $\mathcal{Y}_a$ est placée à droite (*resp.* gauche) de $\mathcal{Y}_b$. Donc les non-arêtes entre x et les sommets de $\mathcal{Y}_b$, orientées de gauche à droite, forcent l'orientation de e de la gauche vers la droite.

Puisque l'algorithme s'arrête lorsque la partition ne contient que des singletons et que chaque étape préserve la propriété A, le résultat produit est bien une extension linéaire de $\overline{G}$ si et seulement si G est un graphe de co-comparabilité. □

Analyse de complexité : Les seuls points à détailler sont le nombre de fois où un sommet est utilisé comme pivot et la complexité nécessaire à retrouver une source dans un module (*i.e.* le premier sommet du module dans un ordre Lex-BFS). Pour plus de détail sur le reste, il suffit de se référer à la preuve de l'algorithme 22.

Commençons par la recherche d'une source dans un module. Ayant calculé un ordre Lex-BFS σ sur les sommets du graphe, nous pouvons considérer la liste des sommets $L(E)$ ordonnée suivant σ. La manière de couper une classe en deux nous garantit que dans chaque sous-classe, l'ordre σ est respecté (pour plus de détails voir le chapitre 5). Nous pouvons donc sélectionner une source en temps constant.

Chaque opération d'affinage se fait en la taille du voisinage d'un sommet. Comptons le nombre de fois qu'un sommet peut être utilisé. Il faut donc analyser la procédure *AjoutPivot*. Soit $\mathcal{X}$ une classe coupée en $\mathcal{X}_a, \mathcal{X}_b$ à une étape donnée. Si $\mathcal{X}$ appartient déjà à $Pivots$, alors nous ajoutons $\mathcal{X}_a$ à $Pivots$ et $\mathcal{X}$ joue le rôle de $\mathcal{X}_b$. Finalement aucun sommet pivot n'a réellement été rajouté. En revanche, supposons que $\mathcal{X}$ n'appartienne pas à $Pivots$. Cette situation se produit quand les sommets de la classe $\mathcal{X}$ ont été utilisés pour affiner la partition et que depuis $\mathcal{X}$ n'a pas été affinée. En insérant la plus petite classe parmi $\mathcal{X}_a$ et $\mathcal{X}_b$, nous assurons qu'un sommet n'est utilisé pour pivoter que si la taille de la classe qui le contient est inférieure à la moitié de la classe qui le contenait la dernière fois qu'il a été utilisé. Donc chaque sommet ne peut être utilisé qu'au plus $\log n$ fois. Nous obtenons donc une complexité globale en $O(n + m \log n)$. □

Théorème 8.22 *L'algorithme 24 calcule une orientation transitive de $\overline{G}$ si et seulement si G est un graphe de co-comparabilité. Sa complexité est $O(n+m\log n)$.*

Remarque 8.23 *Il est très facile d'adapter l'algorithme 24 au calcul d'une extension linéaire de G si et seulement si G est un graphe de comparabilité. Il suffit de modifier la procédure InsertàDroite de la manière suivante.*

Procédure 25: *InsertàDroite* $(\mathcal{X}, \mathcal{X}_a, p)$

Données : deux classes $\mathcal{X}, \mathcal{X}_a$ et un élément p de E
Résultat : *vrai* si $\mathcal{X}_a$ doit être insérer à droite de $\mathcal{X}$, *faux* sinon
début
  soit x un élément de $\mathcal{X}$;
  retourner (x est à droite de p);
fin

8.4 Conclusion

Ce chapitre a permis dans un premier temps déxhiber les différences entre les problèmes liés à la décomposition modulaire et les problèmes abordés au chapitre 7 (partition fonctionnelle maximum, partition relationnelle, minimsation d'automates). Les deux dernières applications sont basées sur un paradigme très intéressant qui, associé à l'affinage de partition, permet d'obtenir des algorithmes simples et efficaces. Enfin, nous avons vu avec l'orientation transitive, qu'un algorithme était très facilement modifiable pour travailler sur le graphe complémentaire sans avoir besoin de le calculer. Cet aspect est généralisable à tous les algorithmes présentés dans cette partie.

Les trois algorithmes présentés font appel à la règle de Hopcroft. C'est leur point faible. Pour toute amélioration en complexité, nous devons être capables de mieux choisir les pivots. La gagure algorithmique consiste alors à préserver la simplicité des algorithmes et des structures de données. Nous étudions maintenant un cas, la reconnaissance des graphes d'intervalles, pour lequel nous sommes capables d'améliorer simplement la règle de choix.

Chapitre 9

Règle astucieuse de choix d'un pivot

Dans ce dernier chapitre, nous nous intéressons à la reconnaissance des graphes d'intervalles puis à la généralisation de ce problème au test des 1 consécutifs d'une matrice booléenne. Pour ces deux applications, nous n'améliorons pas la complexité des algorithmes connus. En revanche, nous proposons un algorithme basé sur l'affinage de partition qui nous semble plus simple que les précédents qui utilisaient soit les PQ-arbres soit la décomposition modulaire.

Nous avons choisi de présenter ces algorithmes dans un chapitre séparé pour deux raisons. La première concerne la règle de choix des pivots. Nous proposons dans un premier temps une version de l'algorithme basée sur la règle de Hopcroft. Puis nous améliorons la complexité en mettant au point une règle de choix plus évoluée. Nous avons déjà mentionné que ce type de modification était possible pour la décomposition modulaire et l'orientation transitive. Mais contrairement à ces applications, la règle que nous proposons est simple à mettre en oeuvre.

La seconde raison concerne la généralisation au test des 1 consécutifs. Nous montrons comment un même algorithme peut travailler sur des représentations différentes des données. Nous avons déjà vu, avec l'orientation transitive, comment il était simple de travailler sur le complémentaire sans le calculer.

9.1 Reconnaissance des graphes d'intervalles

Un graphe $G = (V, E)$ est un *graphe d'intervalles* si et seulement si c'est le graphe d'intersection d'un ensemble d'intervalles sur la droite réelle, *i.e.* : à chaque sommet $x \in V$ est associé un intervalle et deux sommets x et y sont adjacents si et seulement si leurs intervalles respectifs intersectent (voir figure 9.1). Nous nous intéressons à la reconnaissance des graphes d'intervalles.

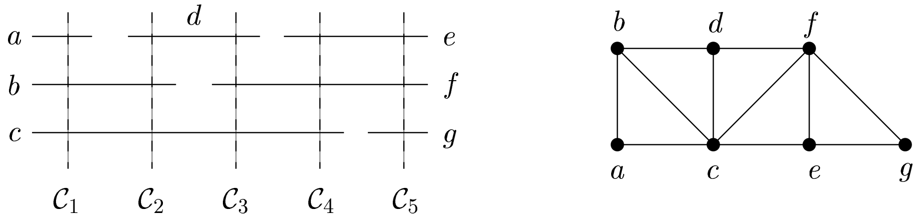


FIG. 9.1 – *Représentation géométrique d'un graphe d'intervalles. La chaîne de cliques,* $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5$ *peut se lire directement sur les intervalles.*

La caractérisation suivante, due à Gilmore et Hoffman, nous servira de base :

Théorème 9.1 *[GH64] Soit un graphe $G = (V, E)$. Les propriétés suivantes sont équivalentes :*

1. *G est un graphe d'intervalles.*
2. *G est un graphe triangulé et son complément est un graphe de comparabilité.*
3. *Les cliques maximales de G peuvent être ordonnées de telle sorte que les cliques maximales contenant un sommet donné x soient consécutives. Un tel ordonnancement est appelé **chaîne de cliques**.*

Etant donnée un chaîne de cliques, nous pouvons en déduire en représentation en intervalles du graphe. A chaque sommet, nous associons l'ensemble des cliques le contenant. Par définition, cet ensemble apparaît consécutivement, c'est donc un intervalle. Puisque pour deux sommets adjacents, il existe une clique maximale les contenant, leurs intervalles intersectent.

Le théorème 9.1 nous indique que le complémentaire d'un graphe d'intervalles est un graphe de comparabilité. A partir d'une chaîne de cliques (ou d'une représentation en intervalles) d'un graphe G, nous pouvons en déduire une extension linéaire du graphe complémentaire $\overline{G}$. Pour chaque sommet x, nous considérons les bornes, $bg(x)$ et $bd(x)$, de son intervalle sur la chaîne. Nous dirons que $x < y$ si et seulement si $bg(x) < bg(y)$ ou bien si $bg(x) = bg(y)$ et $bd(x) < bd(y)$. En cas d'égalité des deux bornes, nous choisirons un ordre arbitraire. En orientant les non-arêtes de la gauche vers la droite, nous obtenons une orientation transitive de $\overline{G}$. Rappelons juste que si (x, y) est une non-arête de G, alors $bd(x) < bg(y)$ (ou l'inverse) et $x < y$ dans l'ordre ci-dessus. La non-arête (x, y) est donc orientée de x vers y. S'il existe une non-arête (x, y), orientée de x vers y et une non-arête (y, z), orientée de y vers z, alors la transitivité de l'ordre implique que (x, z) soit une non-arête orientée de x vers z.

Nous avons donc une correspondance entre une représentation sous forme d'intervalles, une chaîne de cliques et une orientation transitive du complémentaire. De même qu'il existe plusieurs orientations transitives pour un graphe de comparabilité quelconque, il existe plusieurs chaînes de cliques et représentations intervallaires. Dans le cas des graphes premiers, il n'existe qu'une orientation transitive; pour les graphes d'intervalles premiers, il n'existe qu'une chaîne de cliques.

Les algorithmes de reconnaissance

En se basant sur l'équivalence $1 \Leftrightarrow 2$ du théorème 9.1, nous pouvons dorénavant utiliser les algorithmes que nous avons présentés pour reconnaître les graphes d'intervalles. Tout d'abord, en utilisant Lex-BFS [RTL76], nous pouvons tester en temps linéaire si le graphe est triangulé. Ensuite, nous pouvons appliquer l'algorithme d'orientation transitive pour obtenir une extension linéaire du graphe complémentaire en $O(n + m \log n)$. Rappelons que le résultat escompté est obtenu si et seulement si le graphe est un graphe de co-comparabilité. C'est donc le cas pour les graphes d'intervalles. Il faut alors tester le résultat de l'algorithme d'orientation. Mc Connell et Spinrad [MS94] ont montré que pour les graphes d'intervalles, ce test peut être fait en linéaire. Ainsi nous obtenons un algorithme en $O(n + m \log n)$. Notons enfin que Mc Connell et Spinrad ont récemment montré [MS97] que la complexité de l'algorithme d'orientation transitive peut être linéaire. Toutefois, cette amélioration est très compliquée à mettre en oeuvre.

De plus il existe déjà des algorithmes linéaires de reconnaissance des graphes d'intervalles. Ils sont tous basés sur l'équivalence $1 \Leftrightarrow 3$ du théorème 9.1. Le premier est dû à Booth et Lueker en 1976 [BL76]. Cet algorithme fait appel à une structure de données très compliquée mise au point par ses auteurs : les PQ-arbres. C'est un arbre qui permet d'obtenir toutes les chaînes de cliques possibles. Notons que les PQ-arbres sont aussi utilisés pour la reconnaissance des graphes planaires. En 1989, Korte et Möhring ont simplifié cette structure pour le cas des graphes d'intervalles. Cette simplification a été possible en faisant l'observation suivante sur le comportement de Lex-BFS sur les graphes d'intervalles. Ce lemme est à rapprocher du théorème 8.21 :

Lemme 9.2 *[KM89] Soit $\sigma = [x_1, \ldots, x_n]$ un ordre Lex-BFS sur un graphe d'intervalles G. La clique contenant x_1 peut être choisie comme extrémité d'une chaîne de cliques.*

Hsu et Ma ont proposé [HM91] un autre algorithme de reconnaissance plus simple mais très différent. Leur méthode est à rapprocher de celle de Mc Connell et Spinrad pour le calcul d'une extension linéaire d'un graphe de comparabilité. Dans une première étape, ils calculent le sous-graphe premier maximal en faisant de la décomposition modulaire. Dans le cas des graphes triangulés, un algorithme linéaire simple existe. Ensuite étant donné un graphe d'intervalles premier, ils calculent une chaîne de cliques. Rappelons qu'elle est alors unique. L'unicité permet de mener cette seconde étape en temps linéaire.

Plus récemment, Corneil, Olariu et Stewart [COS97b] ont mis au point un algorithme utilisant quatre parcours Lex-BFS avec des règles de choix de pivot particulières pour calculer une orientation transitive de $\overline{G}$ si et seulement si le graphe G est un graphe d'intervalles. Comme nous l'avons déjà mentionné, il est possible de tester ce résultat en temps linéaire.

Nous présentons un algorithme basé sur l'affinage de partition. Pour éviter une étape préliminaire de décomposition modulaire (comme dans l'algorithme de Hsu et Ma), nous utiliserons Lex-BFS de la même manière que pour l'orientation transitive. Nous proposons dans un premier temps un algorithme en $O(n+m\log n)$ qui utilise la règle de Hopcroft pour le choix des pivots. Puis nous montrons comment améliorer cette règle de choix pour trouver systématiquement un pivot qui affine strictement la procédure.

9.1.1 Utilisation de la règle de Hopcroft

Une chaîne de cliques est un ordre total sur les cliques maximales vérifiant une propriété de "consécutivité" par rapport aux sommets. Nous devons donc gérer une partition $\mathcal{P}$ de l'ensemble des cliques maximales du graphe. Puisque les graphes d'intervalles sont des graphes triangulés, leurs cliques maximales peuvent être calculées durant une exécution de Lex-BFS et ordonnées suivant leur ordre de "découverte".

Chaque étape d'affinage consiste à choisir un sommet x et à regrouper dans $\mathcal{P}$ l'ensemble $\mathcal{C}(x)$ des cliques le contenant. Décrivons cette opération :
Soit $\mathcal{X}_1 \in \mathcal{P}$ une classe contenant une clique $\mathcal{C} \in \mathcal{C}(x)$ et une clique $\mathcal{C}' \notin \mathcal{C}(x)$. S'il existe une classe $\mathcal{X}_2$ à droite (resp. gauche) de $\mathcal{X}_1$ contenant une clique de $\mathcal{C}(x)$ alors $\mathcal{X}_1$ sera remplacée par $\mathcal{X}_1 \setminus \mathcal{C}(x), \mathcal{X}_1 \cap \mathcal{C}(x)$ (resp. $\mathcal{X}_1 \cap \mathcal{C}(x), \mathcal{X}_1 \setminus \mathcal{C}(x)$).

En revanche, s'il n'existe pas de seconde classe $\mathcal{X}_2$ contenant une clique de $\mathcal{C}(x)$, nous ne pouvons rien décider. Il faut essayer à nouveau d'affiner la partition avec le pivot x quand $\mathcal{X}_1$ a elle-même été affinée. Dès qu'un sommet x a affiné strictement la partition, alors $\mathcal{C}(x) = \mathcal{Y}_1 \cup \ldots \cup \mathcal{Y}_k$ où $\mathcal{Y}_i$, pour $1 \leqslant i \leqslant k$, est une classe de $\mathcal{P}$. Autrement dit, il ne sera plus jamais nécessaire de pivoter suivant le sommet

x. Nous pouvons donc supprimer ce sommet des cliques de $\mathcal{C}(x)$.

Algorithme 26: Reconnaissance des graphes d'intervalles

Données : Un graphe $G = (V, E)$

Résultat : Une chaîne de cliques $\mathcal{P}$ si et seulement si G est un graphe d'intervalles

début

soit γ l'ordre des cliques maximales de G associé à un ordre Lex-BFS;
$\mathcal{P} \leftarrow (\{\mathcal{C}_1, \ldots, \mathcal{C}_k\})$;
soit $\mathcal{C}(x)$ l'ensemble des cliques contenant x trié suivant γ;
exécuter l'algorithme 9 et la procédure 10 avec les définitions suivantes (les éléments de *Pivots* sont les cliques maximales de G);

fin

Procedure *InitPartition* ($\mathcal{P}$)

si $\mathcal{P}$ *ne contient que des singletons* **alors**
fin du processus et **retourner** $\mathcal{P}$;
soit $\mathcal{C}$ la dernière clique suivant γ d'une classe non-singleton $\mathcal{X}$;
remplacer dans $\mathcal{P}$, $\mathcal{X}$ par $\mathcal{X} \setminus \mathcal{C}, \mathcal{C}$;
ajouter $\mathcal{C}$ à *Pivots*;

Procedure *EnsemblePivot* ($p = (x, \mathcal{C})$)

(x est un sommet de la clique C);
soit $C(x)$ l'ensemble des cliques contenant x;
si $\mathcal{C}(x)$ *est inclus dans une classe* $\mathcal{X}$ *de* $\mathcal{P}$ **alors** **retourner** $\emptyset$;
sinon
supprimer x des cliques de $\mathcal{C}(x)$;
retourner $\mathcal{C}(x)$;

Procedure *InsertàDroite* ($\mathcal{X}, \mathcal{X}_a, p$)

soit p le couple $(x, \mathcal{C})$;
soit $\mathcal{Z}$ la classe de $\mathcal{P}$ contenant $\mathcal{C}$;
retourner ($\mathcal{X}$ est à droite de $\mathcal{Z}$);

Procedure *AjoutPivot* ($\mathcal{X}, \mathcal{X}_a$)

soit $\mathcal{Z}$ la plus petite classe parmi $\mathcal{X}$ et $\mathcal{X}_a$;
si $\mathcal{X} \in$ *Pivots* **alors** ajouter les cliques de $\mathcal{X}_a$ à *Pivots*;
sinon ajouter les cliques de $\mathcal{Z}$ à *Pivots*;

Le lemme 9.2, nous permet de débuter le processus en isolant la dernière clique visitée par un ordre Lex-BFS. Lorsque le processus s'arête, soit toutes les classes sont des singletons et nous obtenons le résultat, soit nous relançons le processus dans les classes non-singletons en utilisant les lemmes 8.20 et 9.2. Nous verrons que, dans ce second cas, une classe $\mathcal{X}$ qui n'est pas un singleton correspond à un module du graphe.

Pivots contient des cliques maximales. A chaque fois qu'une classe $\mathcal{X}$ est coupée en deux lors d'une étape d'affinage, si $\mathcal{X}$ n'est pas déjà dans *Pivots*, alors seules les cliques de la plus petite des deux classes seront ajoutées. Sinon, rajouter l'ensemble de la classe $\mathcal{Y}$ des cliques supprimées de $\mathcal{X}$, n'augmente pas le nombre de pivots. En faisant ainsi, nous appliquons la règle de Hopcroft.

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Une chaîne de cliques est compatible avec $\mathcal{P}$ si et seulement si G est un graphe d'intervalles.*
- ***B:*** *Pour chaque clique d'une classe $\mathcal{X}$, l'ensemble des sommets supprimés est le même.*

Preuve : Soit G un graphe d'intervalle. Supposons que, avant l'étape d'affinage par le sommet x, une chaîne de cliques soit compatible avec $\mathcal{P}$. Si $\mathcal{C}(x)$ ne coupe aucune classe, la propriété A est triviallement vraie. Sinon l'ensemble $\mathcal{C}(x)$ constitue désormais une union de classes continues dans $\mathcal{P}$ et il n'y a qu'une seule manière d'aboutir à ce résultat. Donc la propriété A est préservée par toute étape d'affinage.

Le lemme 9.2 montre que le premier appel à la procédure *InitPartition* produit une partition qui vérifie la propriété A. Montrons que c'est aussi le cas pour chaque appel à la procédure *InitPartition*. Si *Pivots* est vide, alors les classes qui ne sont pas des singletons sont stables pour les ensembles $\mathcal{C}(x)$ ayant strictement affiné la partition $\mathcal{P}$. Autrement dit, pour chaque classe $\mathcal{X}$ qui n'est pas un singleton, l'ensemble des sommets supprimés par l'aglorithme est le même. La propriété B est donc vérifiée. Soit $S(\mathcal{X})$ l'ensemble de ces sommets supprimés pour la classe $\mathcal{X}$. Et soit M l'ensemble des sommets d'une clique de $\mathcal{X}$ n'appartenant pas à $S(\mathcal{X})$. Alors pour tout sommet $y \in M$, $N(y) = S(\mathcal{X})$: autrement dit M est un module. Les lemmes 8.20 et 9.2 montrent que chaque appel à *InitPartition* produit une partition vérifiant la propriété A.

Comme l'algorithme termine lorsque les classes sont des singletons et que chaque étape préserve la propriété A, le résultat est une chaîne de cliques si et seulement si le graphe est un graphe d'intervalles. $\square$

Analyse de complexité : Dans la procédure *InitPartition*, le choix de la dernière clique selon γ peut se faire en temps constant si la liste $L(E)$ contenant les cliques a initialement été triée selon γ (voir chapitre 5). A nouveau la procédure *InsertàDroite* demande un temps constant. La suppression des sommets d'une clique dans *EnsemblePivot* peut aussi se faire en temps constant en manipulant des pointeurs (une marque "utilisée" sur les sommets permet éventuellement de simplifier la structure de données). Puisque la complexité du calcul d'un ensemble pivot S et l'affinage selon S est $O(|S|)$, examinons le nombre de fois où ces procédures sont appelées.

Si la partition a été strictement affinée selon l'ensemble $\mathcal{C}(x)$ des cliques contenant un sommet x, alors elle sera toujours stable pour $\mathcal{C}(x)$. Seulement il arrive qu'un sommet utilisé n'affine pas strictement la partition (c'est le cas quand $\mathcal{C}(x)$ n'est inclus que dans une classe). Il est donc parfois nécessaire de pivoter plusieurs fois sur le même sommet. L'utilisation de la règle de Hopcroft dans la procédure *AjoutPivot* assure qu'une clique est ajoutée au plus $\log n$ fois. Puisqu'un sommet est supprimé dès qu'il a strictement affiné la partition, il sera utilisé au plus $\log n$ fois. La complexité globale est donc $O(n + m \log n)$. □

9.1.2 Un algorithme linéaire

A chaque fois que nous faisons appel à la règle de Hopcroft pour choisir les pivots, cela signifie que nous ne savons pas bien choisir les pivots. Autrement dit, nous pouvons pivoter suivant un ensemble S sans que cet ensemble affine strictement la partition. La règle de *la plus petite moitié* assure que chaque ensemble pivot S est utilisé au plus $\log n$ fois.

Nous avons cité la décomposition modulaire et l'orientation transitive [MS94, MS97] comme deux exemples où il existe une règle de pivot plus judicieuse que la règle de Hopcroft. La complexité peut donc être améliorée en linéaire. En revanche, la mise en oeuvre de ces règles reste trop compliquée pour être implémentée. La reconnaissance des graphes d'intervalles est à notre connaissance la seule application à présenter une règle judicieuse de choix des pivots simple à implémenter. Nous présentons maintenant cette règle.

Arbre de cliques

Trouver un pivot qui affine strictement la partition courante consiste à trouver un sommet qui appartient à plusieurs classes de la partition $\mathcal{P}$: *i.e.* un sommet x dont l'ensemble des cliques $\mathcal{C}(x)$ le contenant n'est pas inclus dans une classe de $\mathcal{P}$. Il ne faut donc pas utiliser un pivot trop tôt.

Au chapitre 1, nous avons étudié les *arbres de cliques* des graphes triangulés. Un arbre de cliques est un arrangement des cliques maximales en un arbre $\mathcal{T}$ tel que l'ensemble des cliques $\mathcal{C}(x)$ contenant un sommet x induise un sous-arbre de $\mathcal{T}$. Puisque les graphes d'intervalles sont des graphes triangulés, ils admettent aussi un arbre de cliques. Clairement une chaîne de cliques est un arbre de cliques particulier. Nous avions vu aussi comment une exécution de Lex-BFS pouvait être adaptée pour construire un arbre de cliques.

Montrons comment utiliser un arbre de cliques pour calculer les pivots. Supposons que nous disposions d'un arbre de cliques $\mathcal{T}$ dont les arêtes sont étiquetées par l'intersection des deux cliques qu'elle connecte. Soit $\mathcal{X}$ un ensemble de cliques maximales connexe dans $\mathcal{T}$. En lisant l'étiquette de l'arête connectant une clique maximale $\mathcal{C} \in \mathcal{X}$ à une clique maximale $\mathcal{C}' \notin \mathcal{X}$, nous obtenons l'ensemble des sommets appartenant à une clique maximale de $\mathcal{X}$ et à une clique maximale extérieure à $\mathcal{X}$ (voir figure 9.2).

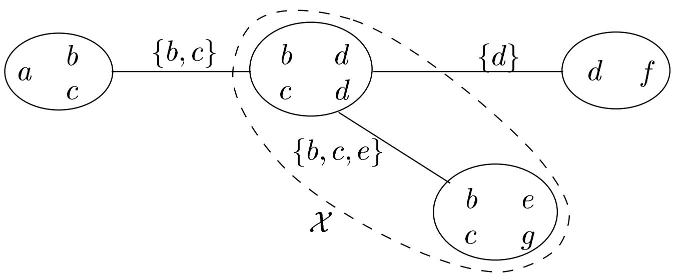


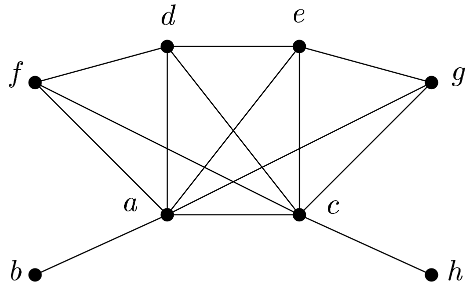
FIG. 9.2 – *Les sommets appartenant à une clique $\mathcal{C} \in \mathcal{X}$ et $\mathcal{C}' \in \mathcal{X}$ sont : b, c, d.*

Mise en oeuvre de la règle

Les éléments de *Pivots* seront désormais des ensembles de sommets correspondant à l'intersection entre deux cliques n'appartenant pas à la même classe. Nous devons donc réécrire la procédure 27.

Procédure 27: *AjoutPivot*$(\mathcal{X}, \mathcal{Y})$

Données : Deux classes $\mathcal{X}$ et $\mathcal{Y}$
début
  pour chaque *arête* $(\mathcal{C}, \mathcal{C}')$ *de l'arbre de cliques où* $\mathcal{C} \in \mathcal{X}$ *et* $\mathcal{C}' \in \mathcal{Y}$ **faire**
    ajouter $\mathcal{C} \cap \mathcal{C}'$ à *Pivots*;
    supprimer $(\mathcal{C}, \mathcal{C}')$ de l'arbre;
fin



Cliques maximales :

$\mathcal{C}_1 = \{a, b\}$

$\mathcal{C}_2 = \{a, c, d, e\}$

$\mathcal{C}_3 = \{a, c, d, f\}$

$\mathcal{C}_4 = \{a, c, e, g\}$

$\mathcal{C}_5 = \{c, h\}$

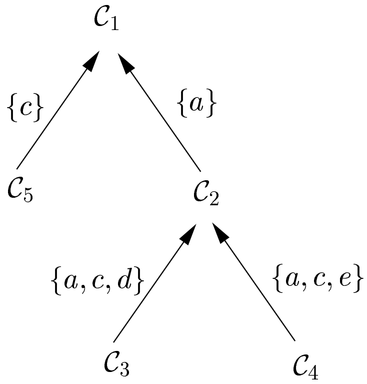
Ordre Lex-BFS sur les sommets :

$\sigma = [h, g, f, e, d, c, b, a]$

Ordre sur les cliques maximales :

$\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5$

Arbres de cliques associé à σ :



Partitionnement des cliques :

$\mathcal{P} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5\}$ (0)

$\mathcal{P} = \{\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}, \{\mathcal{C}_5\}\}$ (1)

$\mathcal{P} = \{\{\mathcal{C}_1\}, \{\mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}, \{\mathcal{C}_5\}\}$ (2)

$\mathcal{P} = \{\mathcal{C}_1\}, \{\mathcal{C}_2, \mathcal{C}_3\}, \{\mathcal{C}_4\}, \{\mathcal{C}_5\}\}$ (3)

$\mathcal{P} = \{\{\mathcal{C}_1\}, \{\mathcal{C}_3\}, \{\mathcal{C}_2\}, \{\mathcal{C}_4\}, \{\mathcal{C}_5\}\}$ (4)

(0) : partition initiale

(1) : isolement de la dernière clique dans Lex-BFS

(2) : $\{c\}$ est l'étiquette de l'arête incidente à $\mathcal{C}_5$ dans l'arbre de cliques
Nous séparons les cliques contenant c des autres

(3) : relance du processus avec la dernière clique parmi $\{\mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$

(4) : $\{a, c, e\}$ est l'étiquette de l'arête incidente à $\mathcal{C}_4$
affinage suivant a et e (le sommet c a déjà été utilisé)

FIG. 9.3 – *Exécution de l'algorithme 26 avec la procédure 27.*

Analyse de complexité : Le problème de la version utilisant la règle de Hopcroft, est que, lorsqu'un sommet est utilisé comme pivot, nous ne sommes pas sûr d'affiner strictement la partition. Aussi chaque sommet peut être utilisé $\log n$ fois. La procédure 27 assure que lorsqu'un ensemble de sommets R est rajouté à *Pivots*, alors tous les sommets de R sont susceptibles d'affiner strictement la partition (les seuls sommets, pour lesquels ce n'est pas le cas, sont les sommets jumeaux d'un sommet déjà utilisé). En effet R est l'intersection de deux cliques $\mathcal{C}, \mathcal{C}'$ n'appartenant pas à la même classe.

A chaque étape d'affinage, seules les classes affinées vont créer de nouveaux ensembles à ajouter à *Pivots*. Soit $\mathcal{X}$ une classe affinée en $\mathcal{X}_1 = \mathcal{X} \cap \mathcal{C}(x)$ et $\mathcal{X}_2 = \mathcal{X} \setminus \mathcal{C}(x)$. Les sommets appartenant à une clique de $\mathcal{X}_1$ et à une clique de $\mathcal{X}_2$ peuvent être trouvés grâce à l'arbre de cliques : il faut considérer les arêtes entre $\mathcal{C}_1 \in \mathcal{X}_1$ et $\mathcal{C}_2 \in \mathcal{X}_2$. Pour cela, nous marquons chaque arête de l'arbre de cliques incidente aux cliques de $\mathcal{C}(x) \cap \mathcal{X}_1$ et $\mathcal{C}(x) \cap \mathcal{X}_2$. Si une arête est marquée deux fois, alors elle relie deux cliques de la même classe. Si elle n'est marquée qu'une fois, elle relie deux cliques de deux classes différentes. Dans ce cas, nous ajoutons l'ensemble correspondant à *Pivots* et nous supprimons l'arête de l'arbre.

A chaque fois q'une arête est marquée, elle est incidente à une clique contenant un sommet x. Pour chaque sommet, il y en a $|\mathcal{C}(x)|$. Or globalement, pour tous les sommets, nous en avons $O(m)$. Donc ce calcul n'augmente pas la complexité globale de l'algorithme. Nous avons vu que chaque sommet est utilisé au plus une fois comme pivot. Donc la complexité de l'algorithme est $O(n+m)$. □

Théorème 9.3 *L'algorithme 26 avec la procédure 27 calcule une chaîne de cliques si et seulement si le graphe en entrée est un graphe d'intervalle. Sa complexité est $O(n+m)$.*

9.2 Propriété des 1 consécutifs

Dans cette section, nous généralisons l'algorithme 26 au *test des 1 consécutifs d'une matrice booléenne* :

Définition 9.4 *Une matrice possède la **propriété des 1 consécutifs** si et seulement si il existe une permutation des colonnes telle que, dans chaque ligne les 1 soient consécutifs. Nous dirons qu'une telle permutation vérifie la **propriété de consécutivité**.*

Le premier algorithme linéaire connu pour la reconnaissance de ces matrices est dû à Booth et Lueker [BL76]. Il utilise aussi les PQ-arbres et est donc relativement compliqué. Récemment, Hsu en a proposé un autre plus simple [Hsu92].

Nous étudions d'abord les liens entre ce problème et la reconnaissance des graphes d'intervalles. Nous verrons alors qu'une adaptation de Lex-BFS nous permettra d'utiliser l'algorithme 26. Nous proposerons cette adaptation. Il s'agit d'une implémentation qui utilise une représentation d'un graphe par ses cliques maximales et non par les listes d'adjacence.

9.2.1 Matrice des cliques

Une colonne $\mathcal{C}$ est confondue avec son ensemble $r(\mathcal{C})$ de lignes contenant un 1 à la colonne $\mathcal{C}$. Nous dirons que $\mathcal{C}$ est maximale dans la matrice M si $r(\mathcal{C})$ est maximal sur l'ensemble des colonnes de M.

Définition 9.5 *Soit G un graphe. La matrice M* ***sommets-cliques maximales*** *de G est une matrice dont les lignes représentent les sommets de G et les colonnes les cliques maximales de G. L'entrée $M[x, \mathcal{C}] = 1$ si et seulement si le sommet x appartient à la clique maximale $\mathcal{C}$.*

Nous dirons qu'une matrice est *conforme* si et seulement si c'est la matrice sommets-cliques maximales d'un graphe G. A partir d'une matrice M, nous définissons le graphe $G(M)$ dont les sommets sont les lignes de M et deux sommets sont adjacents si et seulement si les lignes correspondantes appartiennent à une même colonne. Remarquons que, si les colonnes de M sont toutes maximales, alors M n'est pas nécessairement la matrice sommets-cliques maximales de $G(M)$. Il peut exister des cliques maximales de $G(M)$ qui ne soient pas des colonnes de M.

Le théorème suivant, déjà connu, montre les liens entre les graphes d'intervalles et les matrices ayant la propriétés des 1 consécutifs :

Théorème 9.6 *Soit M un matrice booléenne. Alors $G(M)$ est un graphe d'intervalles et M sa matrice sommets-cliques maximales si et seulement si les colonnes de M sont maximales et M admet la propriété des 1 consécutifs.*

Preuve : Supposons que $G(M)$ soit un graphe d'intervalles et que M soit sa matrice sommets-cliques maximales. Alors il existe une permutation des cliques maximales de $G(M)$ telle que les cliques contenant un sommet donné soient consécutives (*i.e.* une chaîne de cliques). Autrement dit, il existe une permutation des colonnes de M telle que les colonnes contenant une ligne donnée soient consécutives. Donc M vérifie la propriété des 1 consécutifs.

Montrons la réciproque. La propriété des 1 consécutifs est une propriété héréditaire. Soient $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ trois colonnes de M telles que l'ordre $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ vérifie la propriété de consécutivité. Clairement $(\mathcal{C}_1 \cap \mathcal{C}_2) \cup (\mathcal{C}_2 \cap \mathcal{C}_3)$ est contenu dans $\mathcal{C}_2$. De plus la propriété de consécutivité implique que $C_1 \cap \mathcal{C}_3$ est aussi contenu dans $\mathcal{C}_2$. Donc pour tout triplet de colonnes, il en existe toujours une qui contient les intersections de toutes paires d'entre elles. Gilmore (voir le livre de Berge [Ber70]) a montré que cette propriété caractérise les matrices conformes. Donc M est la matrice sommets-cliques maximales de $G(M)$. Toute permutation vérifiant la propriété de consécutivité correspond à une chaîne de cliques de $G(M)$. □

Pour être sûr de travailler sur une matrice conforme, nous allons construire à partir de M une matrice $\widetilde{M}$ qui est la matrice sommets-cliques maximales d'un graphe d'intervalles si et seulement si M vérifie la propriété des 1 consécutifs. Dès lors, en calculant un ordre Lex-BFS sur les cliques de $G(\widetilde{M})$, nous pouvons construire une chaîne de cliques, autrement dit une permutation des colonnes de $\widetilde{M}$ vérifiant la propriété de consécutivité.

Présentons la manière de construire $\widetilde{M}$. Pour être sûr que toutes les colonnes soient maximales, nous ajoutons pour chaque colonne une ligne ayant un unique 1 à la colonne correspondante. Cet ajout n'augmente pas la taille de la matrice et peut se faire en temps linéaire. De plus, cette opération préserve la propriété de consécutivité. Par ailleurs, travailler sur les colonnes d'une matrice M est équivalent aux cliques de $G(M)$.

9.2.2 Lex-BFS

Le problème est que pour obtenir une complexité linéaire, nous ne pouvons pas nous permettre de calculer la représentation sous forme de listes d'adjacence de $G(\widetilde{M})$. Il nous faut donc adapter l'algorithme Lex-BFS à une représentation sous forme de listes des cliques maximales.

L'algorithme 28 présente une adaptation de Lex-BFS à une représentation sous forme de matrice sommets-cliques maximales d'un graphe. Le théorème 9.6 nous garantit que, si une matrice M possède la propriété des 1 consécutifs, alors le graphe $G(M)$ est un graphe d'intervalles. C'est donc aussi un graphe triangulé. Nous faisons l'hypothèse que $G(M)$ est triangulé: *i.e.* l'ordre sur les sommets calculé par l'algorithme 28 est un ordre Lex-BFS si $G(M)$ est triangulé.

Algorithme 28: Lex-BFS sur la matrice d'incidence sommets-cliques

Données : Une matrice booléenne M

Résultat : Si $G(M)$ est triangulé, un ordre Lex-BFS des lignes de M (*i.e.* des sommets de $G(M)$) et un ordre Lex-BFS sur les colonnes $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ (*i.e.* les cliques maximales de $G(M)$

début

$\mathcal{P} \leftarrow (\{\mathcal{C}_1, \ldots, \mathcal{C}_k\})$;
soit $\mathcal{C}(r)$ l'ensemble des colonnes ayant un 1 à la ligne r;
exécuter l'algorithme 9 et la procédure 10 avec les définitions suivantes (les éléments de $Pivots$ sont les colonnes de M);
L'ordre Lex-BFS sur les sommets est l'ordre dans lequel les lignes ont été choisis pour pivoter;

fin

Procedure *InitPartition* ($\mathcal{P}$)

au i-ème appel:
choisir une colonne $\mathcal{C}$ dans la i-ème classe $\mathcal{X}$;
si $|\mathcal{X}| > 1$ **alors** remplacer $\mathcal{X}$ par $\mathcal{X} \setminus \{\mathcal{C}\}, \{\mathcal{C}\}$ dans $\mathcal{P}$;
ajouter $\mathcal{C}$ à $Pivots$;

au k-ème appel:
fin du processus, **retourner** $\mathcal{P}$;

Procedure *EnsemblePivot* ($p = (r, \mathcal{C})$)

(r est une ligne avec entrée 1 de la colonne $\mathcal{C}$);
supprimer la ligne r;
retourner $\mathcal{C}(r)$;

Procedure *InsertàDroite* ($\mathcal{X}, \mathcal{X}_a, p$)

retourner vrai;

Procedure *AjoutPivot* ($\mathcal{X}, \mathcal{X}_a$)

rien

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *Un ordre Lex-BFS des colonnes est compatible avec $\mathcal{P}$.*
- ***B:*** *Toutes les classes sont des singletons.*

Preuve : Nous supposons que $G(M)$ est triangulé. Un ordre Lex-BFS des colonnes de M correspond à l'ordre de visite par Lex-BFS des cliques maximales de $G(M)$.

Nous allons donc montrer que l'ordre dans lequel les lignes ont été utilisées comme pivot est un ordre Lex-BFS sur les sommets de $G(M)$. Dans la suite de la preuve, nous parlerons des sommets de $G(M)$ (*resp.*. des clique maximales) au lieu des lignes (*resp.*. les colonnes) de M. Donc les sommets numérotés sont les lignes ayant été utilisées comme pivot et les cliques visitées sont les colonnes ajoutées à $Pivots$.

Puisque $G(M)$ est triangulé, l'ordre construit est un ordre d'élimination simplicial. Supposons que l'on ait numéroté les sommets $x_r, \ldots, x_{i+1}$; et que l'invariant soit vérifié jusqu'à cette étape. Si ces sommets constituent l'union des cliques maximales $\mathcal{C}_1, \ldots, \mathcal{C}_k$, alors un sommet x non-numéroté appartient à une clique maximale $\mathcal{C}$ distincte de $\mathcal{C}_1, \ldots, \mathcal{C}_k$. De plus, tous les sommets de $\mathcal{C}$ non-numérotés ont le même voisinage parmi $x_r, \ldots, x_{i+1}$ et ce voisinage est un sous-graphe complet (pour plus de détails sur ce point voir chapitre 1). Donc si x a une marque maximum, tous les sommets non-numérotés de $\mathcal{C}$ ont aussi une marque maximum. Puisqu'ils se marquent les uns les autres, ils devront être numérotés de manière consécutive. C'est ce que fait l'algorithme 28. Il faut donc montrer que lorsqu'une nouvelle clique est ajoutée à $Pivots$, c'est une clique contenant des sommets de marque maximum. L'ensemble des sommets numérotés de la nouvelle clique maximale est exactement l'étiquette du nouveau sommet x_i. Or l'algorithme 28 place systématiquement à droite les cliques contenant le sommet pivot. Autrement dit la classe la plus à droite contenant des cliques non-visitées contient les sommets d'étiquette maximum. Le sommet x_i est donc le bon. Enfin le cas de base est trivial : nous pouvons commencer par n'importe quel sommet, donc n'importe quelle clique maximale. □

Analyse de complexité : Le coût des procédures *AjoutPivot*, *InsertàDroite* et *InitPartition* est constant. La complexité globale dépend donc des opérations de calcul des ensembles pivots et des étapes d'affinage. Pour un ensemble S, elle se fait en $O(|S|)$. Comme chaque ligne est utilisée une fois, nous obtenons une complexité en $O(r + c + m)$ où r est le nombre de lignes, c le nombre de colonnes et m le nombre de 1 de la matrice M. □

9.2.3 L'algorithme

Nous avions vu (voir chapitre 1) qu'en utilisant l'algorithme Lex-BFS classique, il est possible de calculer un arbre de cliques. L'algorithme 28 s'adapte exactement de la même manière. Pour adapter l'algorithme 26 au problème des 1 consécutifs, il suffit de modifier sa procédure princaple de la façon suivante :

Algorithme 29: Test des 1 consécutifs

Données : Une matrice booléenne M

Résultat : Une permutation des colonnes de M vérifiant la propriété de consécutivité si et seulement si M a la propriété des 1 consécutifs

début

calculer $\widetilde{M}$ à partir de M;

calculer un ordre γ des colonnes de $\widetilde{M}$ (*i.e.* des cliques de $G(\widetilde{M})$ avec l'algorithme 28 en calculant un arbre de cliques de $G(\widetilde{M})$;

$\mathcal{P} \leftarrow (\{\mathcal{C}_1, \ldots, \mathcal{C}_k\})$;

soit $\mathcal{C}(x)$ l'ensemble des cliques contenant x trié suivant γ;

exécuter l'algorithme 9 et la procédure 10 avec les définitions suivantes (les éléments de *Pivots* sont les cliques maximales de G);

fin

L'algorithme 29 vérifie l'invariant suivant (la preuve est directe avec les résultats déjà vus) :

Invariant : *La partition $\mathcal{P}$ vérifie la propriété A; si $\mathcal{P}$ ne vérifie pas B, alors il existe un pivot dans Pivots qui affine strictement $\mathcal{P}$.*

- ***A:*** *une permutation des colonnes vérifiant la propriété de consécutivité est compatible avec $\mathcal{P}$ si et seulement si M possède la propriété des 1 consécutifs.*
- ***B:*** *Toutes les classes sont des singletons.*

Analyse de complexité : Nous avons déjà détaillé la complexité de toutes les étapes de cet algorithme. Nous obtenons globalement du $O(r + c + m)$ □

Théorème 9.7 *L'algorithme 29 calcule une permutation des colonnes de M vérifiant la propriété de consécutivité si et seulement si M vérifie la propriété des 1 consécutifs. Sa complexité est $O(r + c + m)$.*

Remarque 9.8 *L'algorithme n'a en fait pas besoin de l'étape prélimaire du calcul de $\widetilde{M}$. En effet l'affinage sur les colonnes ajoutées n'a strictement aucun effet sur la partition $\mathcal{P}$. Nous l'avons introduit pour simplifier la preuve de validité (due au théorème 9.6.*

9.3 Reconnaissance des bipartis convexes

Dans cette section, nous présentons brièvement une dernière application. C'est une conséquence directe des algorithmes vus dans ce chapitre. Dans la section précédente, nous avons défini la matrice sommets-cliques maximales d'un graphe. Nous pouvons faire la même chose avec les graphes bipartis :

Définition 9.9 *Soit $G = (V, E)$ un graphe. Le **biparti sommets-cliques maximales** de G est un graphe $B_c(G) = (V, \mathcal{C}, \mathcal{E})$ où $\mathcal{C}$ est l'ensemble des cliques maximales de G et où il existe une arête entre un sommet x et une clique maximale $\mathcal{C}_i$ si et seulement si $x \in \mathcal{C}_i$.*

Les graphes bipartis convexes sont des graphes possédant aussi une propriété de consécutivité. C'est une sous-classe des graphes bipartis triangulés [Gol80]. Elle est intéressante car un certain nombre de problèmes NP-complets sur les bipartis triangulés deviennent polynômiaux sur les bipartis convexes, notamment le nombre de sauts [Gyö84, Lub91].

Définition 9.10 *Un graphe biparti $B = (X, Y, E)$ est dit **convexe** si et seulement si il existe une permutation τ de Y telle que pour tout sommet $x \in X$, les sommets de $N(X)$ sont consécutifs dans τ.*

En utilisant la notion de chaîne de cliques d'un graphe d'intervalles, le lemme suivant est immédiat :

Lemme 9.11 *Soit $G = (V, E)$ un graphe d'intervalles. Alors $B_C(G)$ est un graphe biparti convexe.*

Seulement la réciproque n'est pas systématiquement vraie. Soit $B = (X, Y, E)$ un biparti convexe. Les voisinages des sommets de Y ne sont pas nécessairement maximaux pour l'inclusion. Nous retrouvons le même problème que pour les matrices : nous devions nous assurer (pour les démonstrations) que la matrice soit conforme. La solution choisie était de rajouter une ligne factice par colonne contenant un unique 1 à la colonne correspondante. La même astuce peut être utilisée : nous rajoutons un voisin propre à chaque sommet de Y. Alors le biparti obtenu sera le biparti sommets-cliques maximales d'un graphe d'intervalles.

Pour retrouver exactement la même forme de données que pour le problème des 1 consécutifs, nous représentons le graphe B par les listes d'adjacence des sommets de Y (*i.e.* les cliques maximales du graphe correspondant) et les sommets. Alors nous pouvons réutiliser les algorithmes de la section précédente pour reconnaître les bipartis convexes :

Théorème 9.12 *En utilisant l'algorithme 29, nous pouvons tester en $O(m+n)$ si un graphe biparti est convexe où n est le nombre de sommets et m le nombre d'arêtes.*

9.4 Conclusion

Ce chapitre nous a permis de mettre en évidence deux aspects intéressants des algorithmes de partitionnement.

- Le premier porte sur la règle de choix de pivot. Nous pouvons la plupart du temps mettre au point un premier algorithme qui utilise la règle de Hopcroft. Il est plus simple du point de vue structures de données, mais légèrement moins efficace. Une étude plus appronfondie peut mettre en évidence une règle "intelligente" de choix des pivots et ainsi améliorer la complexité. Nous avons pu constater que les modifications dans l'algorithme précédent restaient alors mineures.
- Le second point porte sur la représentation des données. Il est intéressant de voir qu'un même schéma d'algorithme peut être utilisé pour différentes représentations des données. Nous avions déjà remarqué au chapitre 8, qu'il etait possible de travailler aussi bien sur le graphe que sur le complémentaire.

Conclusions et Perspectives

Bien que cette thèse soit composée de deux parties relativement indépendantes, le parcours Lex-BFS en a été le fil conducteur. La question initiale était de comprendre cet algorithme dans son cadre d'origine (la reconnaissance des graphes triangulés). A cette époque, nous n'aurions jamais imaginé que cette question nous mènerait trois ans plus tard aux algorithmes de partitionnement. J'espère que nous avons fourni une réponse satisfaisante à cette première question. Dans tous les cas, nous avons répondu à d'autres questions. Chaque avancement appelle une nouvelle interrogation, ouvre sur un nouveau domaine de l'algorithmique. Petit à petit, notre champ d'investigation s'élargit à de nouveaux domaines. J'avoue que cet aspect "indéterministe" de l'avancement de la recherche est un des points qui m'ont le plus grisé durant ces trois années.

Revenons un instant sur ce que je crois être les différents apports de cette thèse. Les trois résultats principaux de la première partie sont :

- Un algorithme de reconnaissance des graphes triangulés généralisant Lex-BFS et MCS.
- Un algorithme linéaire de construction d'un arbre de cliques. Cet algorithme a permis la mise au point d'un nouvel algorithme linéaire pour la reconnaissance des graphes d'intervalles.
- Une heuristique linéaire qui calcule le diamètre d'un graphe triangulé à 1 près. Cette même heuristique obtient les mêmes résultats sur les graphes sans triplet astéroïde.

Suite à l'étude des algorithmes de reconnaissance des graphes triangulés, il nous a semblé vraiment comprendre le fonctionnement de Lex-BFS. Pour ses propriétés diamétrales, nous n'en sommes pas encore là. Il suffit de constater la longueur de la preuve sur le diamètre. Nous ne pensons pas avoir trouvé les bons outils, peut-être devrions nous regarder des propriétés de convexité [FJ86]. Un important travail sur Lex-BFS reste donc à faire. Berry et Bordat [BB97] proposent peut-être une voie de recherche intéressante.

Nous pensons que ce parcours simple amènera de nombreux résultats. Il a déjà permis de simplifier les algorithmes d'orientation transitive et de reconnaissance des graphes d'intervalles. Il devrait pouvoir jouer un rôle important dans la décomposition modulaire. Finalement il nous semble important de comprendre quelles sont les propriétés des derniers sommets visités par Lex-BFS. Ou encore quelles sont les propriétés des derniers sommets visités par un i-ème parcours Lex-BFS ? Enfin, il serait aussi intéressant de considérer d'autres parcours que Lex-BFS. Nous pensons par exemple au parcours doublement lexicographique. Sa complexité n'est certes pas linéaire, mais nous l'avons vu, pour des problèmes comme le diamètre, ce n'est pas important.

L'étude de Lex-BFS a aussi permis de se familiariser avec un paradigme algorithmique intéressant pour reconnaître des classes de graphes : une première étape calcule une "structure" caractéristique en supposant que le graphe appartienne à la de graphes considérée; une seconde teste la correspondqnce du résultat avec la structure escomptée. Nous avons utilisé à plusieurs reprise ce paradigme avec les techniques d'affinage de partition : reconnaissance des cographes et des graphes d'intervalles. Séparer la phase de test du reste permet de mieux situer la difficulté d'un problème : pour reconnaître les graphes de comparabilité (voir chapitre 8) nous avons un algorithme linéaire pour la première étape, mais pas pour la seconde.

Faisons un bilan plus détaillé de la seconde partie. Nous avons essayé de faire une présentation la plus large possible des techniques d'affinage de partition. Nous n'avons pas abordé le *double ordonnancement lexicographique d'une matrice* étudié par A. Lubiw [Lub87] et R. Paige et R. Tarjan [PT87]. Il consiste à permuter les lignes et les colonnes d'une matrice booléenne de telle sorte qu'elles soient ordonnées lexicographiquement. Ce travail reste bien entendu à faire et devrait s'intégrer facilement dans cette étude.

Avant de présenter les différentes conséquences de cette technique, la figure 9.4 résume dans un tableau les résultats algorithmiques que nous avons obtenus. Nous y indiquons soit le chapitre où ils sont développés lorsque nous en sommes les auteurs, soit la référence bibliographique. L'affinage de partition ne mène pas systématiquement aux complexités optimales que nous rappelons aussi. En fait, certains algorithmes optimaux peuvent aussi utiliser des techniques de partitionnement (c'est le cas de la décomposition modulaire par exemple), mais ils ne peuvent pas être considérés comme une extension de l'algorithme générique. Dans tous les cas, l'affinage de partition permet d'obtenir de bonnes complexités (au pire un facteur log est ajouté à la complexité optimale).

Problème	**Complexité présentée**	**Réf.**	**Meilleure complexité**	**Réf.**
Calcul des sommets jumeaux d'un graphe	$O(n+m)$	folklore	$O(n+m)$	folklore
Tri lexicographique de chaînes de caractères	$O(m+k)$	[AHU74]	$O(m+k)$	[PT87]
Partition fonctionnelle maximum	$O(n \log n)$	[AHU74]	$O(n \log n)$	[AHU74]
Minimisation d'automates déterministes	$O(nk \log n)$	[Hop71]	$O(nk \log n)$	[Hop71]
Partition relationnelle maximum	$O(m \log n)$	[PT87]	$O(m \log n)$	[PT87]
Partition modulaire maximum	$O(n+m \log n)$	ch. 8	$O(n+m)$	[CH94] [MS94] [DGM97]
Reconnaissance des cographes	$O(n+m \log n)$	ch. 8	$O(n+m)$	[CPS85]
Orientation transitive d'un graphe	$O(n+m \log n)$	ch. 8	$O(n+m)$	[MS97]
Reconnaissance des graphes d'intervalles	$O(n+m)$	ch. 9	$O(n+m)$	[BL76] [KM89] [HM91]
Test des 1 consécutifs d'une matrice	$O(n+m+k)$	ch. 9	$O(n+m+k)$	[BL76]
Reconnaissance des bipartis convexes	$O(n+m)$	ch. 9	$O(n+m)$	[BL76]

FIG. 9.4 – *Performances des algorithmes présentés et complexités connues*

Le schéma d'algorithme que nous avons proposé met bien en évidence le point critique de tous ces algorithmes : toute amélioration de la complexité des algorithmes présentés consiste à trouver une meilleure règle de choix de pivots.

L'exemple de la reconnaissance des graphes d'intervalles montre combien il est simple de changer la règle de choix des pivots dans un algorithme. Nous espérons améliorer de la même manière la décomposition modulaire et l'orientation transitive. Les algorithmes linéaires existants pour ces deux problèmes nous semblent encore trop compliqués pour être programmés.

Grâce à cette unification, nous avons mieux compris les différences et points communs entre chaque problème. Il est intéressant par exemple de mettre en relation la décomposition modulaire et la minimisation d'automates finis déterministes. Pour la décomposition, nous connaissons des algorithmes linéaires alors que la minimisation d'automates n'admet qu'une solution en $O(nk \log n)$. Puisque leurs algorithmes sont basés sur les mêmes techniques, pourquoi ne pourrions nous pas améliorer la complexité de la minimisation des automates?

Il n'est donc pas interdit de conjecturer l'existence d'un algorithme linéaire pour la minimisation des automates finis déterministes. Pour espérer un tel résultat, il faudrait montrer qu'il existe toujours une manière de choisir les états pivots de sorte que chaque état soit utilisé une seule fois. Une telle propriété existe pour la décomposition modulaire. Cette direction constitue donc un premier objectif en vue d'un algorithme linéaire pour la minimisation d'automates.

Deux des applications que nous avons résolues utilisaient à l'origine une structure de données complexe : les *PQ-arbres*. Il s'agit de la reconnaissance des graphes d'intervalles et de la propriété des 1 consécutifs dans une matrice booléenne. Les PQ-arbres permettent de gérer un ensemble de permutations d'un ensemble E. Pour cela, ils contiennent deux types de noeuds internes, les noeuds P et les noeuds Q, indiquant dans quel ordre peuvent être permutés leurs descendants. Les feuilles sont les éléments de E. Nous pouvons considérer qu'un PQ-arbre contient finalement une hiérarchie de partitions compatibles. La partition que nous gérons dans nos algorithmes correspond à la partition induite par les ancêtres immédiats des feuilles. Il existe une troisième application utilisant les PQ-arbres : le test de planarité d'un graphe. Il constitue naturellement un des prochains thèmes que nous souhaitons aborder.

Nous ne gérons dans notre formalisme que l'opération de partition d'une classe. Une autre technique, *l'union-find* permet de gérer la fusion de deux classes [Tar75, Tar79, TL84]. De nombreuses applications utilisent cette technique. Il serait intéressant de proposer un unique formalisme qui permette de regrouper les opérations de partition et de fusion.

Dans une partition, les éléments de deux classes éléments de deux classes distinctes sont indépendants (*i.e.* deux classes ne fusionneront pas ou ne se mélangeront pas). Les seules interactions apparaissent lors des affinages. Ces aspects peuvent être recherchés dans la mise au point d'algorithmes parallèles. La question naturelle concerne donc une éventuelle généralisation de nos algorithmes à des algorithmes parallèles.

Nous terminerons cette conclusion sur l'exemple utilisé au chapitre 4 : *quicksort*. Cet algorithme est sans doute le plus étudié de tous ces algorithmes de partitionnement. Il est connu que sa complexité en moyenne est en $O(n \log n)$ [Sed77]. En revanche dans le pire des cas, nous obtenons $O(n^2)$. Plus récemment, une version randomisée de *quicksort* a été analysée. L'idée ici est de choisir le sommet pivot de manière aléatoire. Une telle mise en oeuvre a l'avantage d'être très simple et sa complexité est en $O(n \log n)$. Il serait intéressant d'étudier aussi des versions randomisées des algorithmes utilisant la règle de Hopcroft pour le choix des pivots dans les algorithmes de partitionnement. De même pour le calcul du diamètre d'un graphe triangulé et d'un graphe sans triplet astéroïde, le nombre de cas où l'heuristique proposée échoue ne nous semble pas si important. Nous avons juste mis en évidence les sous-graphes qui posent problème. Nous n'avons pas du tout considéré les différents ordres Lex-BFS sur ces sous-graphes. Que donnerait une analyse probabiliste de cette heuristique?

Liste des Algorithmes

Bibliographie

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Design and analysis of computer algorithm.* Addisan-Wesley, 1974.

[Aig79] M. Aigner. *Combinatorial Theory*, volume 234 of *A series of Comprehensive Studies in Mathematics.* Springer-Verlag, Berlin, Heidelberg, New York, 1979.

[BB97] A. Berry and J.P Bordat. Separability generalize dirac's theorem. preprint, 1997.

[BBC92] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique.* Masson, 1992.

[BCD94] A. Brandstädt, V.D. Chepoi, and F.F. Dragan. The algorithmic use of hupertree structure and maximum neighbourhood orderings. In *Proceedings of the 20th Inter. Workshop on Graph-Theoretic Concept in Computer Science*, number 903 in LNCS, Berlin, 1994. Springer-Verlag. à paraître dans Discrete Applied Mathematics.

[Ber60] C. Berge. Les problèmes de colorations en théorie des graphes. *Publ. Inst. Statist. Univ. Paris 9*, 1960.

[Ber70] C. Berge. *Graphes et Hypergraphes.* Dunod, 1970.

[BKM95] J. Basch, S. Khanna, and R. Motwani. On diameter verification and boolean matrix multiplication. Technical Report STAN-CS-95-1544, Dpt of Computer Science, Standford University, 1995.

[BL76] K.S. Booth and G.S. Leuker. Testing for the consecutive ones property, interval graphs and graph planarity using pq-tree algorithm. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[BP92] J.R.S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. preprint, 1992.

[Cap96] C. Capelle. *Decomposition de graphes et permutations factorisantes.* PhD thesis, Univ. de Montpellier II, 1996.

[CH94] A. Cournier and M. Habib. A new linear algorithm of modular decomposition. In *Trees in algebra and programming—CAAP 94* (Edinburgh) Lecture Notes in Computer Science, volume 787, pages 68–84, Berlin, 1994. Springer.

[CLR94] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

[Cor] D.J. Corneil. On the diameter of at-free graphs. private communication.

[COS95] D.G. Corneil, S. Olariu, and L. Stewart. Linear time algorithms for dominating pairs in asteroidal triple-free graphs. Technical Report 294–95, university of Toronto, January 1995.

[COS97a] D.G. Corneil, S. Olariu, and L. Stewart. Asteroidal triple-free graphs. *SIAM Journal on Discrete Mathematics*, 10(3):399–431, 1997.

[COS97b] D.G Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm? Extended abstract, 1997.

[CPS85] D.G. Corneil, Y. Pearl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 14(4):926–934, Novenber 1985.

[Dam97] G. Damiand. Quelques propriétés des graphes distance héréditaires. Master's thesis, Université de Montpellier II, 1997. LIRMM.

[DGM97] E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical modular decomposition. In *Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithm*, pages 26–35. Society of Industrial and Applied Mathematics (SIAM), 1997.

[Dir61] G.A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Uni. Hamburg 25*, 1961.

[DNB96] F. Dragan, F. Nicolai, and A. Brandstädt. Lex-bfs orderings and powers of graphs. In *Graph-Theoretic Concepts in Computer Science, WG'96*, volume 1197 of *Lecture Notes in Computer Science*, pages 166–180. WG'96, 1996.

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific Journal Math.*, 15:835–855, 1965.

[FGS] U. Faigle, O. Goecke, and R. Schrader. On church-rosser decomposition in combinatorial structures. Institüt für ökonometrie und operations research. Rheinische Friedrich-Wilhelm-Universität. Bonn.

[FJ86] M. Farber and R.E. Jamison. Convexity on graphs and hypergraphs. *SIAM Journal Alg. Discrete Meth.*, 7(3):433–444, 1986.

[Gal67] T. Gallai. Transitiv orientierbare graphen. In *Acta Math. Acad. Sci. Hungar.*, volume 18, pages 25–66, 1967.

[Gav74] F. Gavril. The intersection graphs of a path in a tree are exactly the chordal graphs. *Journ. Comb. Theory*, 16:47–56, 1974.

[GH64] P.C. Gilmore and A.J. Hoffman. A characterization of comparability graphs and interval graphs. *Canadian Journal of Mathematics*, 16:539–548, 1964.

[GHP95] P. Galinier, M. Habib, and C. Paul. Chordal graphs and their clique graph. In M. Nagl (Ed.), editor, *Graph-Theoretic Concepts in Computer Science, WG'95*, volume 1017 of *Lecture Notes in Computer Science*, pages 358–371, Aachen, Germany, June 1995. 21st Internationnal Workshop WG'95, Springer.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability - A guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[Gol80] M.C. Golumbic. *Algorithms Graph Theory and Perfect Graphs.* Academic Press, New York University, 1980.

[Gyö84] E. Györi. A minimax theorem on intervals. *Journal of Combinatorial Theory Serie B*, 37:1–9, 1984.

[Hab81] M. Habib. *Substitution des structures combinatoires, théorie et algorithmes.* PhD thesis, Université Pierre et Marie Curie (Paris VI), 1981.

[HL89] C.W. Ho and R.C.T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Inform. Process. Lett.*, 31:61–68, 1989.

[HM91] Hsu and Ma. Substitution decomposition on chordal graphs and applications. In *Proceedings of the 2nd ACM-SIGSAM Internationnal Symposium on Symbolic and Algebraic Computation*, number 557 in LNCS. Springer-Verlag, 1991.

[HMPV97] M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoritical Computer Science*, 1997. to appear.

[Hop71] J. Hopcroft. Am $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of machines and computations*, pages 189–196, New York, 1971. Academic Press.

[HPV97] M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata. Technical Report 97131, LIRMM, 1997.

[Hsu92] W.L. Hsu. A simple test for the consecutive ones property. In *LNCS 650*, pages 459–468, 1992.

[KLS91] B. Korte, L. Lovász, and R. Schrader. *Greedoids.* Number 4 in Algorithms and Combinatorics. Springer Verlag, 1991.

[KM89] N. Kortee and R. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. of Computing*, 18:68–81, 1989.

[KS83] P.C. Kanellakis and S.A Smolka. Ccs expression, finite state processes, and three problems of equivalence. In *Proceedings of ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.

[LB92] C.G. Lekkerkerker and J.C Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1692.

[Les84] M. Lesk. *Couplages maximaux et diamètre de graphes.* PhD thesis, Université Pierre et Marie Curie, Paris 6, October 1984.

[Lub87] A. Lubiw. Doubly lexical orderings of matrices. *SIAM Journ. of Algebraic Disc. Meth.*, 17:854–879, 1987.

[Lub91] A. Lubiw. A weighted min-max relation for intervals. *Journal of Combinatorial Theory Serie B*, 53:151–172, 1991.

[Lun90] M. Lundquist. *Zero patterns, chordal graphs and matrix completions.* PhD thesis, Clemson University, 1990.

[Ma] T. Ma. A counter example for simon's algorithm. unpublished.

[McK] Terry A. McKee. How chordal graphs work.

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, 1995.

[MS94] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, VA), pages 536–545, New York, 1994. ACM.

[MS97] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of undirected graphs. In *Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithm.* Society of Industrial and Applied Mathematics (SIAM), 1997.

[Pey86] B. Peyton. *Some applications of clique trees to the solutions of sparse linear systems.* PhD thesis, Clemson University, 1986.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 1957.

[PS85] F.P. Preparata and M.I. Shamos. *Computational geometry - an introduction.* Springer-Verlag, 1985.

[PT87] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[Rei72] E.M. Reingold. On the optimality of somme set algorithms. *Journal of the ACM*, 19:649–659, 1972.

[Ros70] Donald J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32:597–609, 1970.

[RTL76] Donald J. Rose, R. Endre Tarjan, and George S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing*, 5(2):266–283, June 1976.

[Sed77] R. Sedgewick. Quicksort with equal key. *SIAM Journal on Computing*, 1(2):240–267, 1977.

[Sei74] S. Seinsche. On a property of the class of n-colorable graphs. *Journal on Combinatorial Theory (B)*, 16:191–193, 1974.

[Shi84] D.R. Shier. Some aspect of perfect elimination orderings in chordal graphs. *Discrete Applied Mathematics*, 7(3):325–331, 1984.

[Sim91] K. Simon. A new simple linear algorithm to recognize interval graph. In *Workshop on Computational Geometry CG'91*, number 553 in LNCS, pages 289–308. Springer-Verlag, 1991.

[Spi86] J.P. Spinrad. Graph partitioning, 1986.

[Sum73] D.P. Summer. graphs indecomposable with respect to the x-join. *Discrete Mathematics*, 6:281–198, 1973.

[Tar75] R.E. Tarjan. Efficienty of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22:215–225, 1975.

[Tar79] R.E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint set. *Journal of Computer and System Science*, 18(2):110–127, 1979.

[TL84] R.E. Tarjan and J. Van Leeuven. Worst case analysis of set union algorithms. *Journal of ACM*, 31(2):245–281, 1984.

[TY84] R.E. Tarjan and M. Yannakakis. Simple linear algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergaphs. *SIAM Journal of Computing*, 13:566–579, 1984.

Mots Clés

Algorithmique - Graphes - Diamètre - Affinage de partititons
Graphes triangulés - Graphes d'intervalles - Cographes
Décomposition modulaire - Minimisation d'automate - Matrices booléennes

Résumé

Les parcours en largueur lexicographiques ont été introduits en 1976 pour reconnaître les graphes triangulés. Depuis des propriétés intéressantes de ce parcours ont été mises en évidence (notamment pour la reconnaissance des graphes d'intervalles).

Pour mieux comprendre ces parcours, nous les avons, dans un premier temps, étudiés leur application d'origine : la reconnaissance des graphes triangulés. Nous obtenons une généralisation des algorithmes de reconnaissance des graphes triangulés. Nous en déduisons une unique preuve de ces algorithmes.

Nous mettons ensuite en évidence des propriétés diamétrales des parcours en largeur lexicographiques pour deux classes de graphes : les graphes triangulés et les graphes sans triplet astéroïde. Nous montrons que, deux parcours permettent de trouver un sommet dont l'excentricité diffère du diamètre d'au plus 1.

Les parcours en largeur lexicographiques peuvent être compris comme un processus d'affinage de partition. Cette technique est utilisée dans de nombreuses applications : tri de chaînes de caractères, minimisation d'automates finis déterministes, orientation transitive d'un graphe, reconnaissance des graphes d'intervalles... Nous proposons une structure de données et un algorithme générique pour toutes ces applications. Ainsi nous mettons en évidence des liens entre ces problèmes et pouvons espérer de nouveaux résultats algorithmiques.