Quelques algorithmes parallèles et séquentiels de traitement des graphes et applications

Laurent Viennot

Thèse de doctorat de l'université Denis Diderot Paris 7

Spécialité informatique

présentée par

Laurent Viennot

LITP / IBP

pour obtenir le grade de

Docteur de l'université Paris 7

Sujet de la thèse :

Quelques algorithmes parallèles et séquentiels de traitement de graphes et applications

soutenue le 13 décembre 1996 devant le jury composé de

Michel Morvan (directeur de thèse)
Michel Cosnard (rapporteur)
Michel Habib (rapporteur)
Didier Arquès
Robert Cori
Jean-Louis Dornstetter
Jens Gustedt
Daniel Krob

Table des matières

Remerciements

Je tiens tout d'abord à remercier les membre du Jury, à savoir Didier Arquès Robert Cori, Michel Cosnard, Jean-Louis Dornstetter, Jens Gustedt, Michel Habib, et Daniel Krob, qui ont accepté de se pencher sur mes travaux et particulièrement Michel Habib et Michel Cosnard qui ont bien voulu émettre un rapport sur cette thèse dans des délais assez restreints. Je tiens aussi à remercier Michel Morvan qui m'a laissé organiser mes deux années de thèse comme je l'entendais en m'ouvrant néanmoins de nombreuses portes où j'ai souvent pu trouver mon bonheur. Je voudrais aussi remercier les relecteurs qui ont fait que le nombre de fautes restantes est beaucoup moins désastreux qu'il n'aurait été sans l'aimable participation de Isabelle Guerrin, Eric Thierry et Karell Bertet. Je voudrais aussi remercier pour sa bienveillance Madame Lelouette qui nous quitte alors que j'écris ces lignes et qui a illuminé par sa personnalité le laboratoire Litp.

Introduction

L'intrusion des ordinateurs dans nos vies quotidiennes n'est plus aujourd'hui une hypothèse de science fiction mais une réalité désormais bien ancrée. Et si les romanciers de l'avenir ne se sont pas trompés sur l'explosion démographique des puces, ils ont plutôt été optimistes quant à leur utilisation. L'intelligence artificielle ne reste en effet qu'un outil de l'intelligence humaine que chacun est de plus en plus amené à utiliser. Rares sont ceux qui ne tapent pas eux-mêmes leurs textes, les graphistes dessinent maintenant sur des tablettes graphiques, les cinéastes montent leur films de manière virtuelle, les enfants jouent sur les ordinateurs, et on « voyage » sur Internet. Evoquons encore le minitel, les distributeurs de billets et les cartes à puce. Cette constatation montre deux voies à suivre pour améliorer notre qualité de vie dans de telles conditions. D'une part apprendre aux hommes à maîtriser l'outil informatique et d'autre part rendre cet outil le plus maniable possible.

Si la deuxième voie est l'objectif principal de ceux qui ont fait de l'informatique leur métier, la première peut les amener à promouvoir l'informatique de manière plus pragmatique que ne le fait la publicité pour les grandes marques d'ordinateurs. L'algorithmique a sa place dans ces deux points de vue. Elle est un peu à l'informatique ce que la recette est à la cuisine[1] : pour faire un bon plat ou un bon logiciel, il faut un bon cuisinier, mais surtout une bonne recette. L'algorithmicien est tenu par deux objectifs, l'un pratique et l'autre pédagogique : déterminer quels sont les algorithmes les plus efficaces pour résoudre des problèmes et expliquer pourquoi ces algorithmes sont efficaces. Si l'on va plus loin ces deux objectifs conduisent à dégager des techniques algorithmiques générales qui sont source d'efficacité.

Le champ des problèmes que l'on peut résoudre avec l'informatique est incroyablement vaste. Cette thèse se restreint au domaine des graphes. Les graphes permettent de modéliser de nombreuses structures de données apparaissant en informatique telles que les matrices creuses ou les bases de données. Ils permettent aussi de modéliser de nombreux problèmes où des éléments sont reliés entre eux tels que les circuits électroniques ou les réseaux. Ils forment donc un modèle assez général pour que l'étude de leur traitement algorithmique apporte des réponses

1. L'informatique est ici considéré comme un art subtil et précis au même titre que l'art culinaire dans les traditions gastronomiques les plus raffinées.

ou au moins des indications de solution pour de nombreux problèmes particuliers. Les graphes forment de plus un terrain très riche pour l'algorithmicien par les nombreuses propriétés structurelles qu'ils peuvent posséder.

Le point de départ de cette thèse est le traitement des graphes en parallèle. Le parallélisme est un moyen de calcul assez récent où plusieurs processeurs peuvent travailler en même temps sur les mêmes données. L'idée de base est que l'on peu faire plus de travail à plusieurs que tout seul. Cela pose des problèmes d'organisation qui sont très difficiles à résoudre, mais c'est une vision très excitante car elle permet de poser un regard neuf sur l'algorithmique qui ne considérait jusque là les problèmes que séquentiellement, tâche élémentaire après tâche élémentaire. Le but principal du parallélisme à mon sens est de traiter des problèmes très volumineux que ce soit en temps de calcul ou en taille mémoire. Ce qui est primordial n'est pas de résoudre le problème plus vite qu'en séquentiel à tout prix mais avant tout de résoudre le problème. Un problème trop volumineux en mémoire ne pourra pas être traité par un seul ordinateur (séquentiel) car il ne tiendra pas dans sa mémoire, la seule solution pour traiter alors un tel problème est d'utiliser plusieurs ordinateurs avec plusieurs mémoires. Le parallélisme a été rendu concret par la construction de machines dites parallèles spécialement conçues pour le calcul parallèle d'une part et par la mise en réseau de nombreux ordinateurs d'autre part. Le réseau Internet et l'ensemble des ordinateurs qui y sont reliés constituent potentiellement la plus importante de toutes les machines parallèles. Si l'utilisation de toutes ces machines est impossible à centraliser, certains projets, tels que la factorisation de très grands nombres premiers, ont déjà été réalisé grâce au réseau Internet et à la participation de quelques milliers d'utilisateurs qui ont prêté le temps d'inactivité de leur ordinateur. Il y a là une puissance de calcul formidable qu'il serait dommage de ne pas essayer d'utiliser (cela pose bien sûr de nombreux problèmes d'organisation, de propriété privée et même politiques).

Il n'existe pas de modèle général de parallélisme permettant à la fois de concevoir des algorithmes sans se soucier de la machine sur laquelle ils seront implantés et de prédire les performances de l'algorithme une fois implanté[2]. Cette thèse aborde l'algorithmique des graphes par l'étude de problèmes particuliers dans trois modèles différents de parallélisme.

Il existe des problèmes assez généraux de graphe dont les seules parallélisations que l'on connaisse ne sont pas efficaces, ce qui implique que le traitement parallèle de certains problèmes est impossible à l'heure actuelle. Ils sont connus comme ceux que l'on ne sait résoudre en parallèle qu'en calculant la fermeture transitive (ce qui peut être trop coûteux). Un premier pas vers le regroupement de ces problèmes en une classe clairement identifiée est donné dans ce chapitre. Dans cette optique, l'étude de problèmes particuliers dans différents modèles permet de cerner les techniques que l'on peut malgré tout utiliser dans le traitement des

2. Une tentative, le modèle BSP [85], va dans ce sens, mais elle est encore trop récente pour que l'on puisse juger de son succès.

graphes.

Le chapitre 1 est consacré au modèle PRAM qui est le modèle de parallélisme le plus simple qui soit : plusieurs processeurs ont accès à une mémoire partagée. Même avec la simplification apportée par le modèle, certains problèmes restent difficiles à résoudre. On suppose dans ce modèle que le nombre de processeurs est polynômialement borné en la taille du problème dans un premier temps, puis on essaye de borner le nombre de processeurs par le travail du meilleur algorithme séquentiel (pour obtenir un algorithme dit optimal). Le modèle PRAM constitue à mon avis une première approche vers la parallélisation d'un problème, il permet de trouver les idées algorithmiques parallèles qui permettront de résoudre en partie un problème en s'autorisant et en encourageant un maximum de parallélisme. Nous verrons qu'il serait bon de rajouter un troisième temps dans cette approche où l'on supposerait que le nombre de processeurs est borné par la taille du problème. Le paragraphe 1.2 introduit une représentation adaptée au traitement algorithmique des ordres de dimension fixée d et permet de calculer une représentation classique de l'ordre, ce calcul est lié au traitement de requêtes géométriques dans un espace de dimension d. Le paragraphe 1.3 est consacré à la reconnaissance en parallèle des ordres N-free et le paragraphe 1.4 traite de la reconnaissance des graphes de comparabilité. D'une manière générale, l'étude de classes particulières de graphes permet de résoudre des problèmes qui sont difficiles dans le cas général en utilisant une structure algorithmique sous-jacente à la classe considérée. Le problème de la reconnaissance consiste à trouver cette structure.

Le chapitre 2 est au consacré au modèle CGM qui est un modèle de machine parallèle dite « à gros grain » qui privilégie l'étude du placement distribué des données d'un problème, c'est-à-dire sur les différentes mémoires des ordinateurs qui vont travailler ensemble sur le problème. On y fait l'hypothèse que le nombre de processeurs est borné par une fonction de la taille du problème (typiquement la racine carrée). Ce modèle favorise le traitement séquentiel de morceaux du problèmes (en parallèle sur les différents processeurs). Il prend le contre-pied du modèle PRAM et la combinaison des deux approches peut permettre la résolution parallèle concrète d'un problème. Ce chapitre reprend les problèmes abordés dans le modèle PRAM et en fournit des solutions dans le modèle CGM. Le paragraphe 2.4 étudie la portabilité de la représentation introduite pour les ordres de dimension fixée et aborde d'un peu plus près le problème de la réponse à des requêtes dans une base de données. Les paragraphes 2.3 et 2.5 reprennent dans le modèle CGM l'étude de la reconnaissance en parallèle des ordres N-free et des graphes de comparabilité respectivement. Un algorithme de «*list-ranking* » est de plus présenté dans la section 2.2, cet outil intervient dans le calcul des composantes connexes d'un graphe dans ce modèle.

Le chapitre 3 est consacré à un « modèle de calcul » très particulier issu d'un problème de téléphonie GSM. Ce chapitre regroupe d'une part les différentes idées algorithmiques qui s'appliquent à un tel problème soumis à de multiples

contraintes et d'autre part des simulations permettant d'évaluer la pertinence des différentes idées. Ce problème est de nature continue mais on peut néanmoins y apporter des solutions issues de l'algorithmique discrète telles que les techniques liées aux composantes connexes d'un graphe. Par soucis de continuité, un algorithme de composante connexes est donné dans chacun des trois modèles abordés. Le paragraphe 3.2 donne une collection des idées algorithmiques qui peuvent aider à la résolution de ce problème, et le paragraphe 3.3 compare les principaux algorithmes présentés à la section précédente à l'aide des résultats de quelques simulations.

Enfin, le chapitre 4 est consacré à une nouvelle technique algorithmique : l'affinage de partition. Le paragraphe 4.1 tente de cerner cette technique et montre les ressemblances entre différents algorithmes existants. Cette technique nous permettra de généraliser certains de ces algorithmes à la résolution d'autres problèmes proches. L'affinage de partition nous permettra ensuite dans le paragraphe 4.2 de donner des algorithmes simples pour résoudre la reconnaissance des graphes d'intervalles et l'orientation transitive, deux problèmes dont les solution algorithmiques efficaces étaient jusque là très difficiles à implanter et reposaient sur des structures de données complexes.

La section finale de chaque chapitre ne s'appelle pas « Conclusion » de manière à imager un peu plus son contenu, mais elle en a un peu la vocation. Elle est consacrée à quelques problèmes ouverts que la rédaction du chapitre m'a inspiré ou m'a fait voir sous un autre angle.

Cette thèse est le fruit de deux années et demi de recherche, ce qui signifie d'une certaine manière une grande quête mystique solitaire mais aussi beaucoup de travail en équipe. Les différents travaux présentés ici ont fait ou vont faire l'objet de publications dont la liste est donnée ci-dessous. Seuls les algorithmes et les théorèmes qui sont le fruit d'un de ces travaux ne comportent pas de citation. Cette liste me permet de plus de citer les amis avec qui j'ai eu le plaisir de travailler. Il manque toutefois le nom de Michael Bender avec qui j'ai beaucoup travaillé sur le problème de la barrière de la fermeture transitive sans aboutir au moindre résultat tangible, ainsi va la recherche.

Chapitre 1

- A compact data structure and parallel algorithms for permutation graphs
  Jens Gustedt, Michel Morvan, Laurent Viennot WG '95 (LNCS 1017)
- Parallel N-free order recognition
  Laurent Viennot TCS 2412
- Parallel comparability graph recognition and modular decomposition
  Michel Morvan et Laurent Viennot STACS '96 (LNCS 1046)
- Un algorithme d'enracinement d'arbre
  Laurent Viennot en cours de rédaction

- Structure de donnee compacte permettant pour les ordres de dimension d, utilisation dans le modèle PRAM et dans le modèle CGM
  Jens Gustedt, Michel Morvan, Laurent Viennot en cours de rédaction

Chapitre 2

- Un algorithme parallèle de list-ranking probabiliste dans le modèle CGM
  Laurent Viennot en cours de rédaction
- Un algorithme de reconnaissance des graphes de comparabilité dans le modèle CGM
  Laurent Viennot en cours de rédaction

Chapitre 3

- Collaboration avec Nortel Matra Cellular
  Jean-Louis Dornstetter, Daniel Krob, Michel Morvan, et Laurent Viennot
  non publié pour raisons de confidentialité
  Partie graphique du simulateur par Stéphane Gosne

Chapitre 4

- Lex-BFS a Partition Refining Technique, Application to Transitive Orientatin and Consecutive 1's testing
  Michel Habib, Christophe Paul, Laurent Viennot soumis
- Quelques algorithmes linéaires de reconnaissance autour de Lex-BFS
  Christophe Paul et Laurent Viennot soumis

Notations

Les termes utilisés dans ce texte sont principalement ceux de la théorie des ensembles. Il faut garder à l'esprit qu'ils ont toujours une traduction informatique, un ensemble est par exemple souvent géré avec une liste doublement chaînée. Les détails de gestion d'ensembles avec des structures de données informatiques faisant partie des rudiments de l'algorithmique, ils seront souvent omis. Les structures de données permettant de représenter les graphes seront rapidement discutées à la fin de cette section ; encore une fois, les différentes structures de données possibles se déduisent directement des différentes définitions ensemblistes possibles et des différentes manières de représenter un ensemble.

Remarquons que l'inverse est aussi vrai : de nombreuses structures de données informatiques se traduisent en termes de graphes. Le schéma général d'enregistrements (les «*record* » en pascal) et de pointeurs représente un graphe orienté où les sommets sont les enregistrements et les arcs sont les pointeurs. Pour obtenir des algorithmes efficaces, il est important d'utiliser au maximum les propriétés théoriques que peut avoir ce graphe. Une matrice représente elle aussi un graphe dont les arcs sont étiquetés. Toute la différence entre l'algorithmique des graphes et celle des matrices vient du fait que l'on « oublie » les entrées nulles de la matrice. Aussi de nombreux algorithmes de graphes sont reliés au calcul sur les matrices creuses.

Une liste presque exhaustive des termes utilisés en algorithmique des graphes va maintenant suivre. Les figures 2 et 3 en illustrent la plupart. Je conseille au lecteur pressé de ne consulter que les figures et les paragraphes concernant les ensembles et les représentations informatiques des graphes.

Ensembles

Un *ensemble* A est une collection d'*éléments* $x \in A$ qui sont généralement numérotés quand ils sont stocké sur un ordinateur (ne serait-ce que par une adresse mémoire). En particulier, on exprime presque toujours la complexité des algorithmes en fonction du *cardinal* $|A|$ des ensembles A qu'ils manient. On notera $\subseteq$, $\subset$, $\cap$, $\cup$, $+$ et $-$ les opérateurs classiques d'*inclusion*, d'inclusion *stricte*, d'*intersection*, d'*union*, d'union *disjointe* et de *différence ensembliste*. Le calcul informatique du résultat de ces opérateurs se fait généralement en triant les ensembles considérés selon un ordre total.

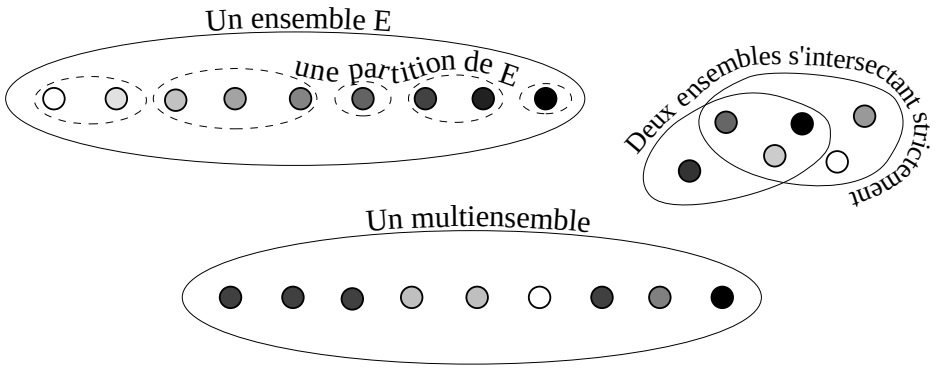


FIG. 1 – *Exemples d'ensembles.*

Si un ensemble de référence E est donné (ce qui sera souvent le cas dans cet ouvrage avec $E = V$ ou $E = V^2$), le complémentaire d'un sous-ensemble $A \subseteq E$ sera noté $\overline{A} = E - A$. On dira que deux ensembles s'*intersectent* («*overlap* » en anglais) si leur intersection n'est pas vide, et qu'ils s'intersectent *strictement* si de plus aucun n'est inclus dans l'autre. Des sous-ensembles $A_1, \ldots, A_k$ forment une *partition* de E s'ils vérifient $E = A_1 + \cdots + A_k$. Les A_i seront alors souvent appelés des *classes*. Une partition est généralement représentée en associant à chaque classe un numéro et à chaque sommet le numéro de sa classe.

Dans un *multi-ensemble* un même élément peut apparaître plusieurs fois[3]. C'est une notion plus proche de l'informatique, comparer par exemple l'opération d'union avec des ensembles et avec des multi-ensembles.

Graphes

Dans un graphe généralement noté $G = (V, \mathcal{E})$, les *sommets* qui sont les éléments de V sont reliés par des *arêtes* qui sont les éléments de $\mathcal{E} \subseteq V^2$. Dans un graphe *non orienté*, les arêtes $\widehat{uv}$ dont les *extrémités* sont u et v, sont en général représentées par la présence redondante dans la structure de données des couples $(u, v) \in \mathcal{E}$ et $(v, u) \in \mathcal{E}$. On dira aussi que u et v sont *reliés* par $\widehat{uv}$ ou que u et v sont *voisins* (ou *adjacents*). L'ensemble noté $N(u)$ des voisins de u s'appelle le *voisinage* de u. Dans un graphe *orienté*, les arêtes uv sont orientées de leur *origine* u vers leur *destination* v. On ne parle plus alors d'arête mais d'*arc* uv. On dira aussi que uv est un arc *sortant* de u et un arc *entrant* de v.

Une arête $\widehat{uv}$ peut être orientée par l'arc uv ou par l'arc *inverse* vu. L'ensemble des arcs obtenu en inversant l'orientation de chaque arc d'un graphe orienté est noté $\mathcal{E}^{-1}$. En pratique, un graphe non orienté est souvent représenté sous forme

3. En informatique, on peut toujours distinguer des éléments identiques (par leur adresse mémoire par exemple).

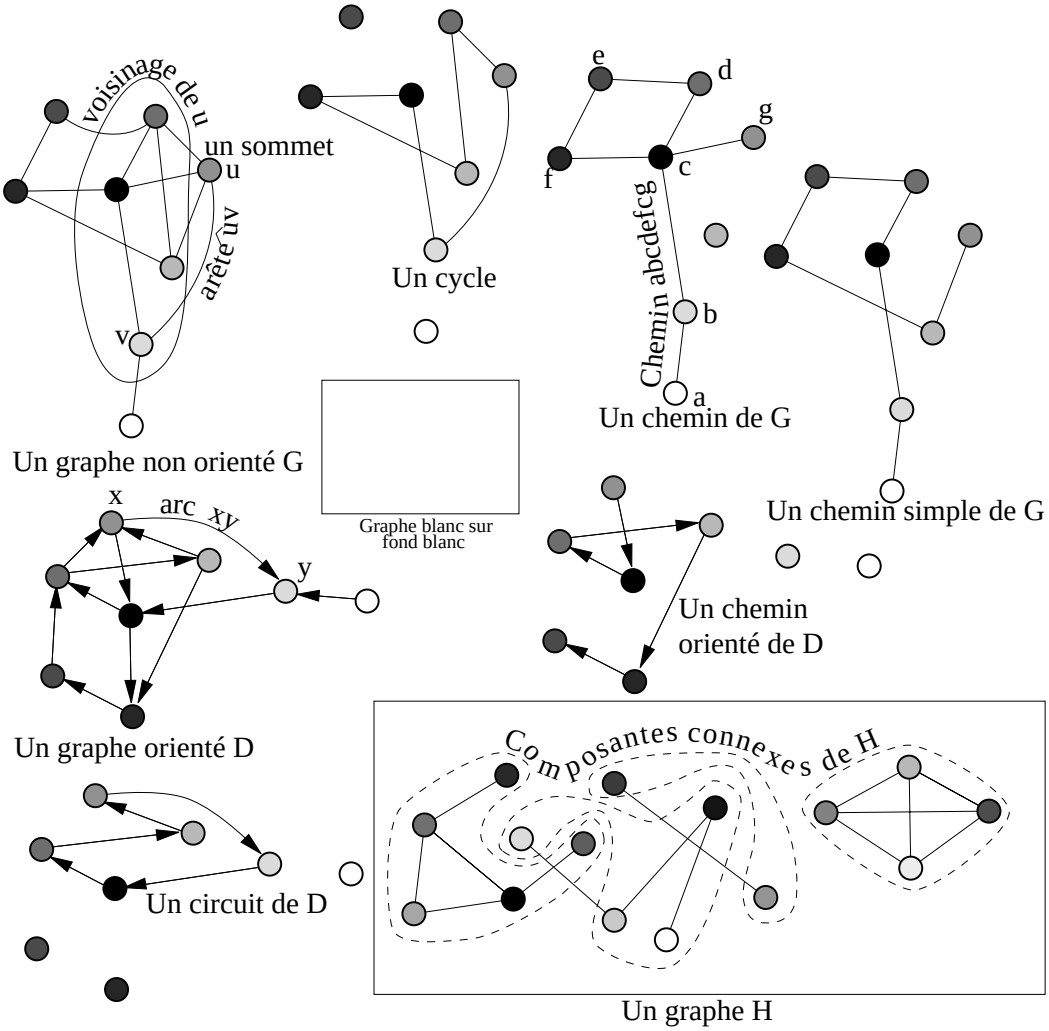


FIG. 2 – *Exemples de graphes. Remarquer que D est une orientation (non transitive) de G.*

d'un graphe orienté *symétrique*, c'est-à-dire tel que $\mathcal{E} = \widehat{\mathcal{E}}$ où $\widehat{\mathcal{E}}$ est le *symétrisé* de $\mathcal{E}$.

Si $M \subseteq V$ est un sous-ensemble de sommets, on notera $\mathcal{E}_M$ l'ensemble des arcs (ou des arêtes selon la nature du graphe considéré) dont les deux extrémités sont dans M et on notera $G_M = (M, \mathcal{E}_M)$ le *sous-graphe induit* par M. Si $\mathcal{A} \subseteq \mathcal{E}$ est un sous-ensemble d'arcs (ou d'arêtes), on en notera $V_\mathcal{A}$ l'ensemble des extrémités et $G_\mathcal{A} = (V_\mathcal{A}, \mathcal{A})$ le sous-graphe induit par $\mathcal{A}$. Les lettres majuscules droites telles que M désigneront en général des ensembles de sommets et les lettres majuscules rondes telles que $\mathcal{M}$ des ensembles d'arêtes ou d'arcs. Dans un graphe non orienté, un *module* M est un ensemble de sommets se comportant tous de la même façon vis à vis des autres sommets, plus formellement : pour tout $u \in V - M$, u est

relié soit à tous les sommets de M, soit à aucun.

Dans le cas où $\mathcal{E}$ est un multi-ensemble, on parlera d'arcs ou d'arêtes *multiples*. D'une manière générale les graphes considérés dans cet ouvrage sont supposés sans *boucles*, c'est-à-dire qu'un sommet n'est jamais relié à lui-même.

Un *chemin* est une suite de sommets où chacun est relié au suivant. Un chemin *orienté* est une suite de sommets où il sort de chaque sommet un arc vers le suivant. Un chemin est *simple* s'il ne *passe* qu'une fois au plus par sommet, le premier sommet pouvant toutefois être confondu avec le dernier. On appelle *cycle* ou (*circuit* dans le cas orienté) un chemin dont les deux extrémités sont confondues. Un graphe est *connexe* si et seulement s'il existe un chemin reliant toute paire de sommets. Les *composantes connexes* d'un graphe quelconque sont les sous-graphes connexes maximaux pour l'inclusion, ou de manière équivalente les sous-ensembles de sommets recouverts par ces sous-graphes.

Le jargon utilisé en algorithmique des graphes se place à cheval sur l'informatique et les mathématiques, aussi la définition des termes n'en est pas complètement rigoureuse. Je prie les bourbakistes de bien vouloir me pardonner de privilégier l'intuition à la rigueur.

Ordres

Un *ordre* partiel ou plus simplement ordre, est une relation irréflexive et transitive. On dit aussi ordre *partiel*. Les ordres les plus populaires sont les ordres *totaux* (où un sommet est toujours plus petit ou plus grand que n'importe quel autre) et les arbres dont je parlerai un peu plus loin. Remarquons aussi qu'une matrice triangulaire induit un ordre par ses entrées non nulles.

Une relation étant un graphe, un ordre est un graphe orienté $P = (V, <)$, en cas d'ambiguïté l'ensemble des arcs est noté $<_P$. On note habituellement $u < v$ pour $uv \in <$ et on dit alors que u *précède* v et que v *succède* à u. On ne parlera pas du voisinage de u mais de l'ensemble $Succ(u)$ de ses *successeurs* et de l'ensemble $Pred(v)$ de ses *prédécesseurs*. Les *maxima* de l'ordre sont les sommets qui n'ont aucun arc sortant et les *minima* sont ceux qui n'ont aucun arc entrant. Un minimum est encore appelé une *source* et un maximum un *puits*.

Un graphe orienté est un ordre si et seulement s'il est sans circuit, et *fermé transitivement* c'est-à-dire vérifiant $u < v$ et $v < w \Longrightarrow u < w$, ou encore pour tout chemin orienté, il existe un arc de l'origine du chemin vers sa destination (l'irréflexivité interdit donc les circuits).

A l'inverse, tout *graphe orienté sans circuit* $G = (V, \mathcal{E})$ représente un ordre unique P appelé sa *fermeture transitive* qui est obtenu en rajoutant tous les arcs reliant l'origine d'un chemin orienté à sa destination. Un graphe est dit sans circuit quand il ne contient aucun circuit. Un graphe est sans circuit si et seulement s'il possède un *tri topologique*, c'est-à-dire un parcours des sommets où l'origine de tout arc est visitée avant sa destination ou encore un ordre total $L = (V, <_L)$ tel que $u <_P v \Longrightarrow u <_L v$ soit encore $<_P \subseteq <_L$. L est aussi appelée *extension*

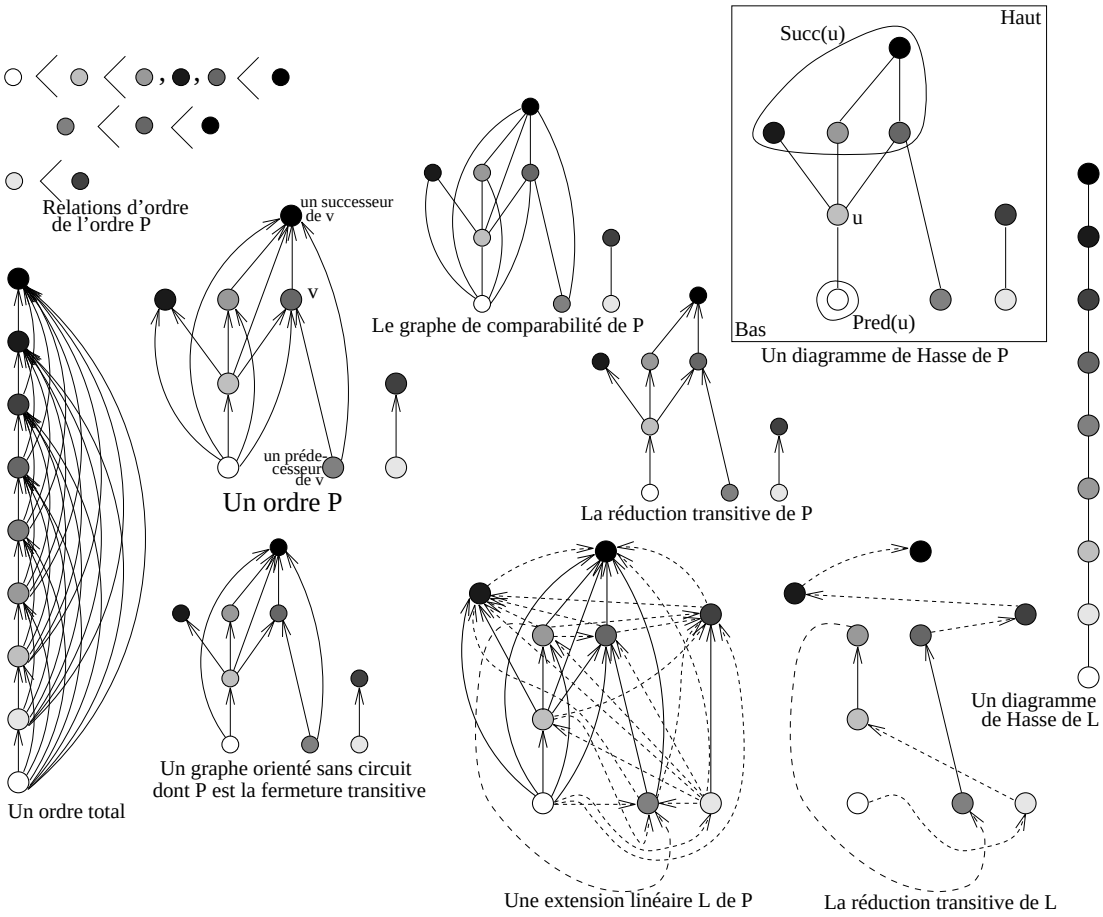


FIG. 3 – *Exemples d'ordres.*

linéaire de P.

En revanche, un ordre est représenté par plusieurs graphes orientés sans circuit, mais un seul d'entre eux est minimal pour l'inclusion des ensembles d'arcs, il s'appelle la *réduction transitive* de tout graphe orienté sans circuit représentant l'ordre. Les arcs de la réduction transitive sont appelés *arcs de couverture*, ils forment l'ensemble minimal d'arcs permettant de représenter l'ordre. Si uv est un arc de couverture, on dira que u précède *immédiatement* v et que v succède *immédiatement* u. On note $ImSucc(u)$ l'ensemble de ses successeurs *immédiats* et $ImPred(v)$ l'ensemble de ses prédécesseurs immédiats. On dessine généralement un ordre en représentant son *diagramme de* HASSE qui est un dessin de sa réduction transitive où l'on omet les orientation des arêtes qui sont implicitement orientées du bas vers le haut.

Remarquons que tout sous-graphe d'un graphe orienté sans circuit induit par un sous-ensemble de sommets ou un sous-ensemble d'arcs est encore un graphe orienté sans circuit et que tout sous-graphe d'un ordre induit par un ensemble de

sommets est encore un ordre.

Le *graphe de comparabilité* d'un ordre P est son symétrisé, c'est-à-dire le graphe obtenu en oubliant l'orientation des arêtes de P. $\widehat{uv}$ est une arête du graphe de comparabilité de P si et seulement si u est *comparable* avec v, ce qui signifie $u <_P v$ ou $v <_P u$. Un graphe est un graphe de comparabilité s'il existe une orientation de ses arêtes qui fournit un ordre.

Arbres et forêts

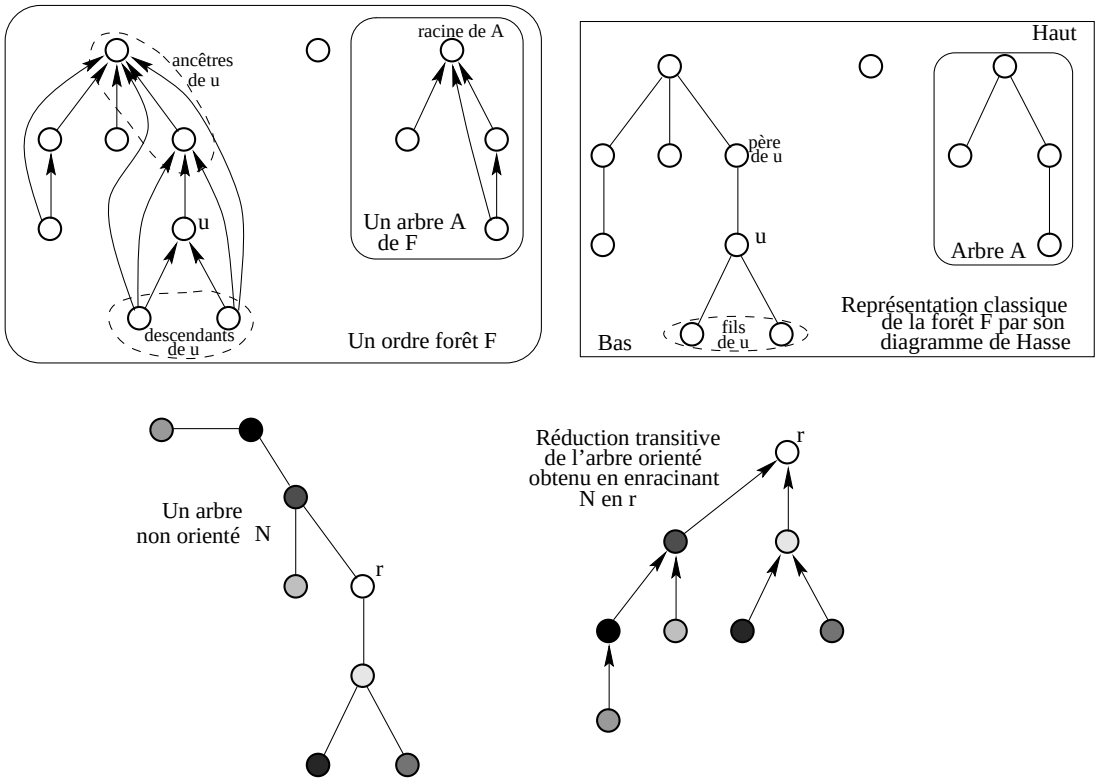


FIG. 4 – *Exemples d'arbres et de forêt.*

Une *forêt* est un ordre où l'ensemble des successeurs de chaque sommet u est totalement ordonné. Les successeurs d'un sommet sont appelés *ancêtres* et les prédécesseurs *descendants*. L'élément minimal parmi les ancêtres d'un sommet u s'appelle le *père* de u, l'élément maximal s'appelle la *racine*. Les sommets ayant même racine induisent un sous-graphe appelé *arbre* qui est constitué d'une racine et de ses descendants [4]. Les racines sont les maxima de la forêt, les minima sont appelés *feuilles*. Un sommet u et ses descendants induisent un *sous-arbre* de

4. Les arbres sont les composantes connexes de la forêt.

racine u. La réduction transitive d'une forêt est donnée par les arcs reliant chaque sommet à son père.

Un arbre non orienté est le symétrisé de la réduction transitive d'un arbre, c'est-à-dire un graphe non orienté connexe et sans cycle bien sûr. *Enraciner* un arbre non orienté consiste à en orienter les arêtes de manière à obtenir la réduction transitive d'un arbre.

L'utilité des arbres en informatique n'est plus à démontrer.

Représentations informatiques des graphes

Dans tout cette thèse, en l'absence d'ambiguïté, n désignera le nombre de sommets du graphe considéré et m le nombre de ses arcs. Les complexités des algorithmes seront données en fonction de n et m puisque la taille de l'entrée se déduit elle-même de n et m selon la structure de données utilisée. Les sommets correspondent toujours à quelque objet en mémoire, en particulier, ils sont au moins numérotés par leur adresse. La représentation en machine d'un graphe tient donc dans la représentation de ses arcs qui est un sous-ensemble de V^2.

Il y a principalement deux manières de représenter un sous-ensemble en machine : soit en stockant une variable booléenne pour chaque élément qui est mise à 1 si l'élément est dans le sous-ensemble et à 0 sinon, soit en donnant la liste exhaustive de ses éléments. D'autre part, l'ensemble des arcs peut être considéré soit dans sa globalité, soit comme l'union des voisinages de chaque sommet. En combinant ces alternatives, on obtient quatre structures de données possibles pour représenter un graphe $G = (V, \mathcal{E})$:

- une *matrice d'adjacence* M $n \times n$ de booléens telle que $uv \in \mathcal{E}$ si et seulement si $M[u, v] = 1$,
- une *liste des arcs* composée des couples (u, v) tels que $uv \in \mathcal{E}$,
- les *listes d'adjacence* de chaque sommet u, chacune composée de la liste des v tels que $v \in N(u)$,
- les tableaux d'adjacence de chaque sommet u, chaque tableau T_u étant composé de n variables booléennes telles que $T_u[v] = 1$ si et seulement si $v \in N(u)$, ce qui ressemble fort à une matrice d'adjacence.

Dans la pratique, c'est souvent les listes d'adjacence qui apparaîssent naturellement (dans le cas des enregistrements reliés par des pointeurs par exemple). Les deux autres structures de données apparaîssent plutôt avec les graphes obtenus en considérant les entrées non nulles d'une matrice. Les matrices peuvent être représentées en Maple sous forme de listes des entrées non nulles avec les tables (ou en déclarant `M:=array(sparse,...);`).

On peut passer d'une représentation à l'autre avec un travail linéaire (en $O(n+m)$ ou $O(n^2)$ selon les cas). Que ce soit en séquentiel ou en parallèle, le seul passage qui pose un problème est celui de la liste d'arcs à une des deux autres représentations car il exige de trier les arcs selon leur origine. Cela ne pose pas de

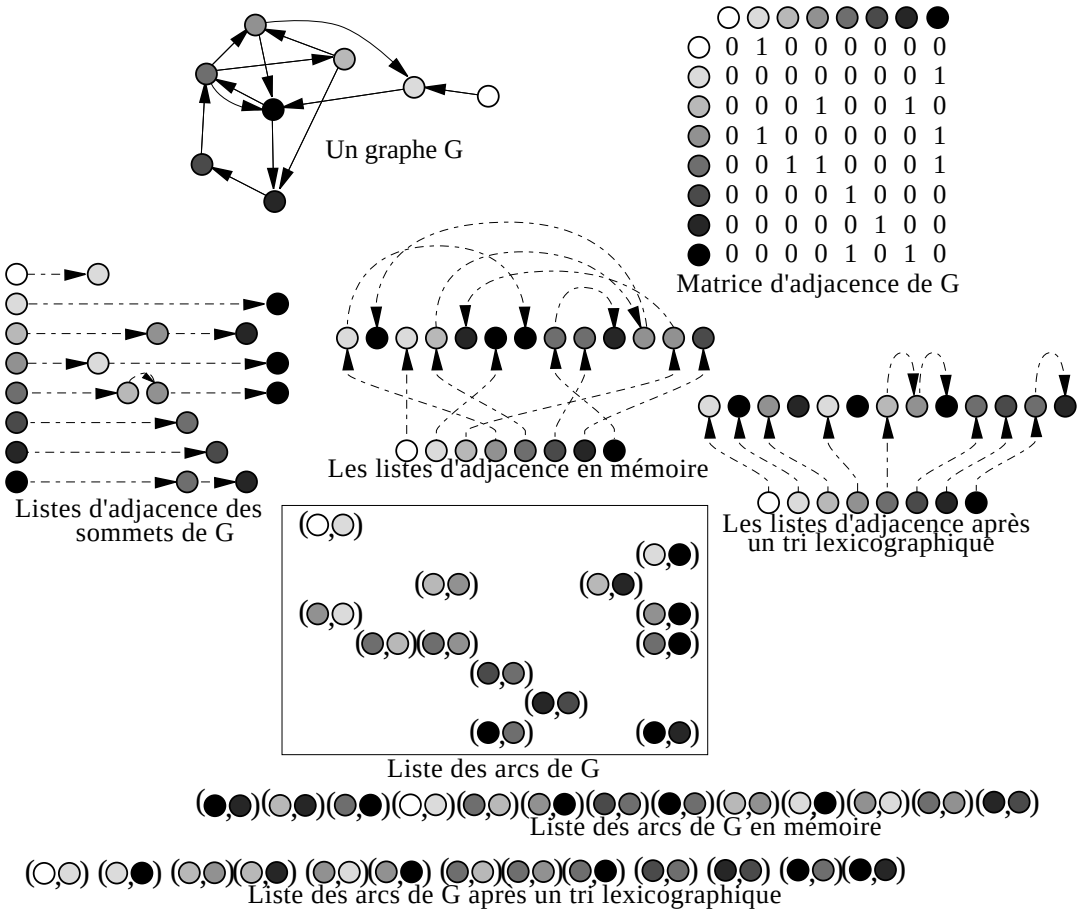


FIG. 5 – *Différentes représentations d'un graphe.*

problème dans tout modèle où le tri du géomètre (ou «*bucket-sort* » en anglais) est linéaire.

En algorithmique des graphes où l'on manie des suites de nombres entre 0 et n qui sont totalement ordonnés par l'ordre $<_{ent}$ sur les entiers, il est souvent intéressant de considérer l'ordre *lexicographique* $<_{lex}$, c'est-à-dire l'ordre $<_{lex}$ du dictionnaire:

$$u_1 \cdots u_k <_{lex} v_1 \cdots v_l \quad \text{si et seulement s'il existe } i \text{ tel que } \begin{cases} u_1 = v_1 \\ \vdots \\ u_i = v_i \\ u_{i+1} <_{ent} v_{i+1} \end{cases}$$

On peut aussi considérer l'ordre *anti-lexicographique* $<_{anti}$ donné par: $u_1 \cdots u_k <_{anti} v_1 \cdots v_l$ si et seulement si $u_k \cdots u_1 <_{lex} v_l \cdots v_1$. Pour des couples, cela donne

$$uv <_{anti} xy \quad \text{si et seulement si} \quad \begin{cases} v <_{ent} y \\ \text{ou bien } v = y \text{ et } u < x \end{cases}$$

Dans le cas où il peut y avoir des arêtes multiples, la matrice d'adjacence doit avoir des entrées de type entier.

Dans le cas de la matrice d'adjacence ou de la liste des arcs, les sommets doivent être numérotés de 0 à $n-1$. Remarquons que le traitement d'un graphe de 2^{64} sommets dépasse largement les capacités de n'importe quel ordinateur. 2^{32} bits représentent environ 500 méga-octets, les graphes dont le nombre de sommets ne tient pas sur 32 bits sont à la limite de ce que l'on peut espérer traiter avec une imposante machine parallèle, et encore faut-il qu'ils aient peu d'arêtes (m de l'ordre de n). On peut donc supposer que le numéro d'un sommet occupe un nombre constant de cases mémoire (une sur les ordinateurs actuels, voire deux).

Chapitre 1

Modèle à mémoire partagée PRAM

Le modèle PRAM a été beaucoup critiqué en raison de sa trop grande simplicité qui le rendrait trop irréaliste. C'est cette simplicité, je crois, qui donne de l'attrait à ce modèle. En effet, les problèmes d'algorithmique parallèle sont extrêmement difficiles. De nombreux problèmes qui paraissent presque anodins en séquentiel n'ont pas de solution parallèle efficace même dans le modèle simplifié qu'est la PRAM (voir le paragraphe 1.5).

Je considère donc un algorithme PRAM comme un premier pas vers la résolution d'un problème en parallèle et nullement comme un aboutissement. Remarquons que la plupart des algorithmes donnés dans le modèle CGM qui est reconnu comme plus proche des machines réelles (et qui effectivement fournit des algorithmes presque directement implantables) sont inspirés en partie de leur homologue PRAM. Ce n'est pas un hasard. Le modèle PRAM a la propriété suivante : tout algorithme écrit pour un certain nombre de processeurs tourne aussi bien avec moins de processeurs[1]. A travail constant, ce modèle pousse donc à obtenir une borne en nombre de processeurs la plus grande possible, à « paralléliser » au maximum, et dans une trop forte mesure en quelque sorte. Pour obtenir un bon programme distribué il faut aussi effectuer localement des phases de calcul séquentiel.

D'autre part il existe certains problèmes dits *P-complets* que l'on ne pas résoudre plus efficacement en parallèle qu'en séquentiel. Ce sont un peu les analogues des problèmes NP-complets en séquentiel. Le modèle PRAM permet de les identifier et de les écarter.

L'algorithmique parallèle des graphes et plus particulièrement des graphes orientés s'avère très difficile car des outils de base en algorithmique séquentielle des graphes sont P-complets (la recherche en profondeur par exemple). Il faut

1. Dans la réalité, quand on a moins de processeurs, on peut faire mieux que simuler un algorithme écrit pour plus de processeurs.

donc trouver de nouveaux outils pour traiter cette structure de données très générale que constitue un graphe. Le présent chapitre s'inscrit dans cette optique. Nous verrons trois problèmes d'algorithmique des graphes et différentes solutions parallèles pour chacun d'eux.

D'autre part, n'ayant toujours pas d'outils parallèles généraux pour remplacer ceux que l'on utilise en séquentiel, l'étude de problèmes particuliers permet de voir comment des propriétés structurelles supplémentaires se marient avec le parallélisme. Cette étude peut nous mener à creuser celles qui sont appropriées au traitement parallèle, pour en trouver d'autres qui s'appliquent de manière plus générale. Par exemple, une question naturelle consiste à se demander ce qui pourrait remplacer la structure d'arbre de recherche en profondeur, ne serait-ce que dans certains problèmes, et dans quels cas est-elle utile?

Dans cet ordre d'idées, nous nous poserons deux problèmes de reconnaissance de classes particulières de graphes. Il est logique qu'il soit plus facile de trouver dans un graphe des structures qui vérifient plus de propriétés que des structures plus générales. En résolvant ces problèmes plus simples, on peut espérer trouver des techniques qui se généraliseront, et en tous cas acquérir une intuition dans ce domaine.

Nous commencerons par présenter le modèle PRAM et les principaux outils parallèles existants. Ces outils existent dans la plupart des machines parallèles mis à part le calcul des composantes connexes que nous verrons donc un peu plus en détail. Les chapitres 2 et 3 reprendront certaines idées de cet algorithme.

Nous verrons dans le paragraphe 1.2 comment appréhender en parallèle la représentation d'un ordre sous forme d'un plongement dans un espace de dimension fixée. Ce problème a des applications en géométrie et dans le traitement de recherches dans des bases de données . Nous introduirons une structure de données qui permet de simplifier la résolution de ce problème. Elle est à mi-chemin entre la structure donnée par le plongement et les représentations classiques des graphes.

Les paragraphes 1.3 et 1.4 sont consacrés à des problèmes de reconnaissance de classes de graphes particuliers. Nous verrons d'abord comment trouver la structure de diagramme d'arcs qui est propre aux ordres N-free, en détectant sa non existence éventuelle. Les ordres N-free sont utilisés comme outil de modélisation et d'analyse de projets. Nous aborderons ensuite la reconnaissance des graphes de comparabilité ainsi que le problème connexe du calcul d'une orientation transitive. Ces problèmes connaissent de récents progrès en algorithmique séquentielle des graphes (voir le chapitre 4) et la présente étude constitue un premier pas vers l'étude de ces nouvelles techniques séquentielles dans l'univers parallèle.

1.1 Présentation du modèle PRAM

Le modèle PRAM qui est l'acronyme de «*Parallel Random Access Memory* » est une simplification à l'extrême des machines parallèles. Dans ce modèle, on considère que plusieurs processeurs ont accès à une mémoire partagée et on y compte le temps de calcul en nombre de lectures et d'écritures dans cette mémoire. Ceci est équivalent au modèle distribué où des processeurs possédant chacun leur propre mémoire communiquent via un réseau avec un temps de communication ne dépendant pas de l'emplacement des processeurs dans le réseau et où le travail final est compté par le nombre total de communications.

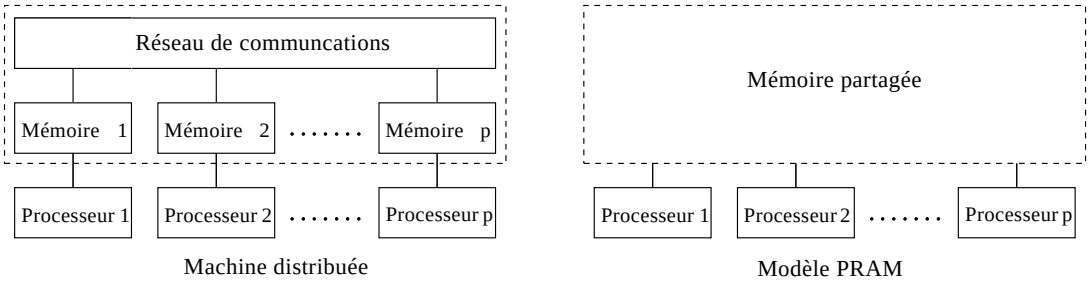


FIG. 1.1 – *La simplification du modèle* PRAM.

Ce modèle est souvent décrié sous prétexte qu'il ne tient pas compte des communications alors qu'au contraire il ne prend qu'elles en compte et dans une trop forte mesure. La simplification du modèle consiste à supposer le coût de l'envoi du contenu de k cases mémoires indépendant de la position dans le réseau des processeurs communiquants et proportionnel à k. Si la première hypothèse est tout à fait justifiée puisqu'elle est plutôt réaliste en ce qui concerne les machines parallèles de la dernière génération, la deuxième est très discutable. En effet, dans la plupart des machines parallèles, le coût d'une communication dépend peu de la quantité de données envoyées dans la mesure où les messages sont généralement courts. D'autre part, l'absence d'une mémoire locale ne permet pas de tirer profit de l'exécution plus rapide de tâches séquentielles effectuées localement (avec un accès à la mémoire plus rapide). Pour cette raison, ce modèle ne pose pas le problème de la répartition des données. Nous verrons dans le chapitre 2 comment le modèle CGM comble ces deux lacunes.

Il est évident qu'un algorithme PRAM ne peut pas s'implanter directement sur une machine réelle. En revanche, il y a deux raisons pour commencer par écrire un algorithme dans le modèle PRAM. Tout d'abord, la plupart des algorithmes parallèles sont basés sur des routines « élémentaires » telles que le tri par exemple, qui font presque toujours partie des routines de haut niveau incluses dans les machines parallèles et implantées au mieux par les fabricants. D'autre part, si l'on ne trouve pas d'algorithme PRAM pour résoudre un problème, il est

plus difficile à priori d'en trouver un dans un modèle plus compliqué. Aussi, ce modèle est utile pour défricher le champs des problèmes que l'on sait résoudre en séquentiel[2] en distinguant ceux qui ont une chance d'être parallélisés (c'est-à-dire qui pourront être résolus en parallèle lorsque leur taille est trop importante pour qu'une machine séquentielle puisse les résoudre) des autres. De plus, il se trouve qu'en pratique, les algorithmes PRAM donnent une bonne base de départ pour produire des algorithmes dans des modèles plus réalistes tels que CGM ou BSP, ou même pour une implantation directe comme certains des algorithmes présentés ici qui s'appuient uniquement sur des routines élémentaires. Disons que le modèle PRAM permet de mettre en exergue le parallélisme inhérent à un problème.

On distingue plusieurs variantes dans le modèle PRAM selon la gestion des accès concurrents à la mémoire. Si le modèle autorise les lectures concurrentes, il est qualifié de CR (pour «*concurrent read* ») et de ER (pour «*exclusive read* ») sinon. Si le modèle autorise les écritures concurrentes, il est qualifié de CW (pour «*concurrent write* ») et de EW (pour «*exclusive write* ») sinon. On ne s'intéresse en général qu'aux modèles EREW PRAM, CRCW PRAM et parfois CREW PRAM. On distingue à nouveau plusieurs cas de CRCW PRAM selon le résultat d'une écriture concurrentielle. Dans la CRCW PRAM *classique*, tous les processeurs écrivant sur un même emplacement doivent écrire la même chose. Dans la CRCW PRAM *arbitraire*, un seul processeur arbitraire réussit à écrire, alors que dans la CRCW PRAM *à priorité*, seul le processeur de plus grand numéro réussit à écrire.

Ces différentes variantes ont un sens physique pour certains prototypes de machines parallèles tels que les étoiles optiques («*optical stars* » en anglais) qui permettent les lectures concurrentes. Cela serait aussi le cas dans un réseau de communication radio par exemple. Dans un réseau de communication, le sens physique que l'on peut intuitivement avoir sur la question (il semble plus difficile de lire à plusieurs un même emplacement mémoire plutôt que chacun un emplacement différent) est respecté puisque le modèle le plus facile à simuler est la EREW PRAM car les lectures ou les écritures concurrentes reviennent à dupliquer les messages. La variante la plus difficile à simuler est à priori la CRCW PRAM à priorité pour laquelle il faudrait trier tous les messages qui arrivent à un processeur selon le numéro de l'émetteur.

D'un point de vue théorique, on peut simuler les différents modèles PRAM les uns avec les autres [67, 69]. Une PRAM à p processeurs peut être simulée par p/p' pas d'une PRAM à $p' \leq p$ processeur (avec un espace mémoire de même taille). Un pas d'une CRCW PRAM (classique, arbitraire, ou à priorité) à p processeurs et m emplacements de mémoire partagée peut être simulé par une EREW PRAM à p processeurs et mp emplacements de mémoire partagée en $O(\log p)$ pas.

Le nombre maximal d'accès à la mémoire partagée d'un processeur lors de

2. Tout algorithme parallèle a une traduction séquentielle directe, ce n'est donc pas la peine d'essayer de résoudre en parallèle des problèmes non résolus en séquentiel, on peut au mieux réussir à paralléliser le meilleur algorithme séquentiel.

l'exécution d'un algorithme sur une PRAM à p processeurs est appelé *temps* d'exécution de l'algorithme et le produit pt est appelé *travail* de l'algorithme [3]. A chaque pas, on considère que tout processeur fait une lecture ou une écriture en mémoire, le travail est donc une borne sur le nombre total d'accès à la mémoire partagée. Tout algorithme tournant en temps t avec p processeurs d'une PRAM peut être simulé par une PRAM à $q \leq p$ processeurs en temps tp/q. Pour cette raison, on essaye toujours à travail constant d'avoir des algorithmes utilisant un nombre maximal de processeurs, ou ce qui revient au même un temps minimal. Le nombre de processeurs n'a pas grande signification puisqu'il est fixe dans une machine réelle ; il sert simplement à indiquer le travail, et on peut d'ailleurs l'omettre si l'on préfère donner le travail à la place. Remarquons qu'une machine séquentielle permet de simuler tout algorithme PRAM en temps pt. Un algorithme est dit optimal si son travail est asymptotiquement le même que le meilleur algorithme séquentiel résolvant le même problème.

Routines « élémentaires »

Voici les outils les plus couramment utilisés dans le modèle PRAM.

Sommes préfixées

Etant donné un tableau de n éléments $a_1, \ldots, a_n$, une EREW PRAM permet de calculer les sommes partielles $a_1 \oplus \cdots \oplus a_i$ pour tout i entre 1 et n en temps $O(\log n)$ avec un travail $O(n)$ [64] (l'algorithme utilise donc $n/\log n$ processeurs). $\oplus$ peut être n'importe quelle opération associative. On parle aussi de calcul préfixé.

List-ranking

Etant donnée une liste de n éléments $a_1, a_{S(1)}, \ldots, a_{S^{n-1}(1)}$ où $S(i)$ désigne le numéro de l'élément suivant le i^{e}, une EREW PRAM permet de calculer les sommes partielles $a_i \oplus a_{S(i)} \oplus \cdots \oplus a_{S^{n-i}(i)}$ en temps $O(\log n)$ avec un travail $O(n)$ [12]. $\oplus$ peut être n'importe quelle opération associative, cet algorithme se généralise au cas où S donne le père de chaque nœud d'un arbre enraciné.

Tri

Etant donné un tableau de n éléments, une EREW PRAM permet de le trier en temps $O(\log n)$ avec un travail $O(n \log n)$. Les trois outils qui viennent d'être présentés ont des solutions parallèles optimales. Comme nous trierons souvent des entiers entre 0 et n^k où $k = 1, 2$ ou 3, citons encore un algorithme de tri du géomètre parallèle [42] qui permet de trier de tels nombres en temps $O(\log n)$ avec un travail $O(n \log\log n)$ dans le modèle CRCW PRAM à priorité.

Nous allons enfin voir un dernier outil parallèle évolué, fondamental en algorithmique des graphes : le calcul des composantes connexes d'un graphe. Comme le problème des composantes connexe peut être considéré comme un modèle adé-

3. Le travail est souvent défini comme le nombre total d'opérations, on simplifiera ici en supposant qu'un processeur fait toujours quelque chose.

quat du problème de synchronisation abordé au chapitre 3, il servira de fil conducteur entre les différents modèles parallèles abordés dans cette thèse.

Algorithme des composantes connexes

L'algorithme [52, 69] qui est largement inspiré des algorithmes d'« union-find », est pensé dans le modèle CRCW PRAM arbitraire. Il consiste à faire grandir les arbres d'une forêts où les sommets d'un même arbre sont toujours dans la même composante connexe. Pour chaque sommet v, $P\grave{e}re(v)$ est son père dans cette forêt (les racines sont leur propre père). Au début de l'algorithme, $P\grave{e}re(v) = v$ pour tout v, et à la fin, chaque arbre correspondra à une composante connexe entière et sera de plus une *étoile* (voir figure 1.2), c'est-à-dire un arbre où tous les sommets sont fils de la racine. Deux sommets u et v seront donc dans la même composante connexe si et seulement si $P\grave{e}re(u) = P\grave{e}re(v)$. Voir l'algorithme 1.1.

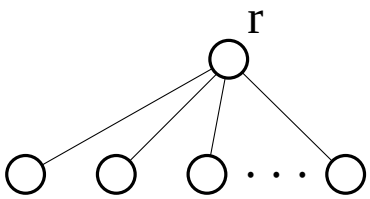


FIG. 1.2 – *Une étoile de racine r.*

Cette algorithme s'exécute en temps $O(\log n)$ sur $n + m$ processeurs d'une CRCW PRAM. Il existe des versions plus compliquées basées sur le même principe qui permettent d'obtenir un travail presque optimal [13].

Arbre couvrant

Si on marque toutes les arêtes qui ont réussi à accrocher une étoile à un autre arbre à un moment de l'algorithme, l'ensemble des arêtes d'une composante connexe forment un arbre non orienté couvrant.

Arbre couvrant de poids maximal

Pour obtenir un arbre couvrant de poids maximal dans chaque composante connexe, il suffit de s'arranger pour que l'arête qui réussit à accrocher une étoile soit de poids maximal parmi les arêtes qui sortent de l'ensemble des sommets de l'étoile. Cela peut se faire en triant à chaque boucle, ce qui conduit à un temps d'exécution total de $O(\log^2 n)$, ou bien avec un temps de $O(\log n)$ en utilisant le modèle plus puissant de CRCW PRAM à priorité où la i^{e} arête dans la liste triée des arêtes est associée au processeur de numéro i.

Arbre couvrant enraciné

Quand on construit un arbre couvrant dans chaque composante connexe, en orientant de plus les arêtes marquées du sommet u qui accroche son étoile vers le sommet v de l'étoile sur laquelle il l'accroche et en inversant les orientations

Algorithme 1.1 [**52, 69**] Composantes connexes

Données : Un graphe $G = (V, \mathcal{E})$.
Résultat : Les composantes connexes de G.
Début
  Pour tout *sommet* v **effectuer** $Père(v) \leftarrow v$
  Répéter $\log_{3/2} n$ *fois :*
    Etape 1 *Accrochage conditionnel*
      Pour tout *arête* uv *ou* vu *dans* $\mathcal{E}$ **effectuer**
        Si u *est dans une étoile* {*voir la procédure de calcul des étoiles*} *et* $Père(u) < Père(v)$ **Alors** $Père(Père(u)) \leftarrow Père(v)$
      {*La condition d'ordre évite de créer des circuits.*}
    Etape 2 *Accrochage inconditionnel*
      Pour tout *arête* uv *ou* vu *dans* $\mathcal{E}$ **effectuer**
        Si u *est encore dans une étoile et* $Père(u) \neq Père(v)$ **Alors** $Père(Père(u)) \leftarrow Père(v)$ {*La première passe est un peu spéciale, il faut remplacer la condition par « u n'a pas été accroché et personne ne s'est accroché sur lui ». A partir de la deuxième passe, les étoiles ont toutes hauteur* 1.}
      {*Les étoiles restantes sont forcément accrochées à des non étoiles, aucun circuit ne peut donc être créé. Après cette étape, tous les arbres ont une hauteur supérieure à* 2.}
    Etape 3 *Contraction*
      {*Les opérations d'accrochage n'ont pas augmenté la somme des hauteurs des étoiles de la composante connexe.*}
      Pour tout *sommet* u **effectuer** $Père(u) \leftarrow Père(Père(u))$ {*Cette opération dite de «*pointer jumping *» divise la somme des hauteurs des arbres d'une composante connexe par un facteur* 3/2 *au moins.*}
  {*La somme des hauteurs des arbres d'une composante connexe étant bornée par le nombre de sommets qu'elle contient, chaque composante n'est constituée à la fin de l'algorithme que d'une seule étoile.*}
Fin
Procédure *calcul des étoiles*
Pour tout *sommet* u **effectuer**
  $Etoile(u) \leftarrow$ Vrai
  Si $Père(Père(u)) \neq Père(u)$ **Alors**
    $Etoile(u) \leftarrow$ Faux
    $Etoile(Père(u)) \leftarrow$ Faux
    $Etoile(Père(Père(u))) \leftarrow$ Faux

des arêtes de l'arbre entre u et la racine r de l'arbre couvrant son étoile à chaque accrochage, on obtient finalement un enracinement de l'arbre couvrant de chaque composante connexe.

Ce problème qui semble anodin n'apparaît pas dans la littérature. Pourtant, il permet de résoudre une généralisation en non orienté en quelque sorte du problème du list-ranking :

> Etant donné un graphe qui est un chemin, en numéroter consécutivement ses sommets.

On peut identifier le chemin de u à r par un list-ranking sur l'arbre. Cela conduit à un algorithme en temps $O(\log^2 n)$ avec un travail $O(n \log n)$. On peut s'arranger pour ne retourner les arêtes qu'une fois l'algorithme de composantes connexes terminé. Ceci permet de conserver la même complexité.

La racine d'un arbre est toujours la racine de l'étoile associée. Conservons donc pour chaque sommet u d'une étoile à l'itération t le numéro $r_t(u)$ de la racine de l'étoile ($0 \leq t \leq \log_{3/2} n - 1$). Dans l'arbre couvrant avec les orientations obtenues sans faire les retournements, marquons les sommets non racine de l'arbre en cours qui ont accroché une étoile de la composante sur une autre. On associe à chaque sommet une étiquette de $\log_{3/2} n$ bits, nulle au début de l'algorithme. Si un sommet u accroche à l'itération t une étoile de la composante sur celle d'un sommet v, le $2t+1^{\text{e}}$ bit de son étiquette est mis à 1. Quand l'étoile est accrochée à une autre à l'instant $t' > t$ par un sommet w, il faut savoir de quel côté de u est w. Si $r_t(w) = r_t(u)$ alors c'est du côté du vieil arbre de u sinon c'est du côté de celui sur lequel il s'est accroché et dans ce dernier cas, le $2t'^{\text{e}}$ bit de l'étiquette de u est mis à 1. Comme un seul sommet accroche une étoile donnée à un instant donné, les étiquettes induisent un ordre total sur les sommets marqués.

Transformons l'arbre couvrant avec les orientations obtenues sans faire les retournements en forêt orientée de la manière suivante. Dupliquons les sommets u marqués pour chaque arc sortant de u. Le nombre d'arcs de l'arbre étant borné par le nombre de nœuds, il n'y a pas plus de n copies de sommets au total. Si u possède un arc entrant, il est raccroché à n'importe laquelle des copies. (Voir la figure 1.3.) Dans chaque arbre mis à part celui qui n'a pas été accroché (et sur lequel tous les autres se sont accrochés), il faut retourner les arcs entre le sommet d'étiquette maximale et la racine. Pour chaque nœud des arbres, on peut calculer l'étiquette maximale parmi celles de ses descendants avec un list-ranking, et par là même identifier les arcs à retourner. On obtient ensuite l'arbre couvrant enraciné en identifiant à nouveaux les diverses copies de chaque sommet marqué.

Nous allons commencer par étudier un problème de calcul de représentation classique d'un ordre à partir d'une représentation particulière. La résolution de ce problème repose sur des techniques similaires à celles utilisées dans le tri «*quicksort* ». Les premiers algorithmes proposés sont simples et fournissent une introduction aux algorithmes PRAM plus compliqués qui suivront. Leur présentation suit la description classique de «*quicksort* » mais nous verrons au chapitre 4

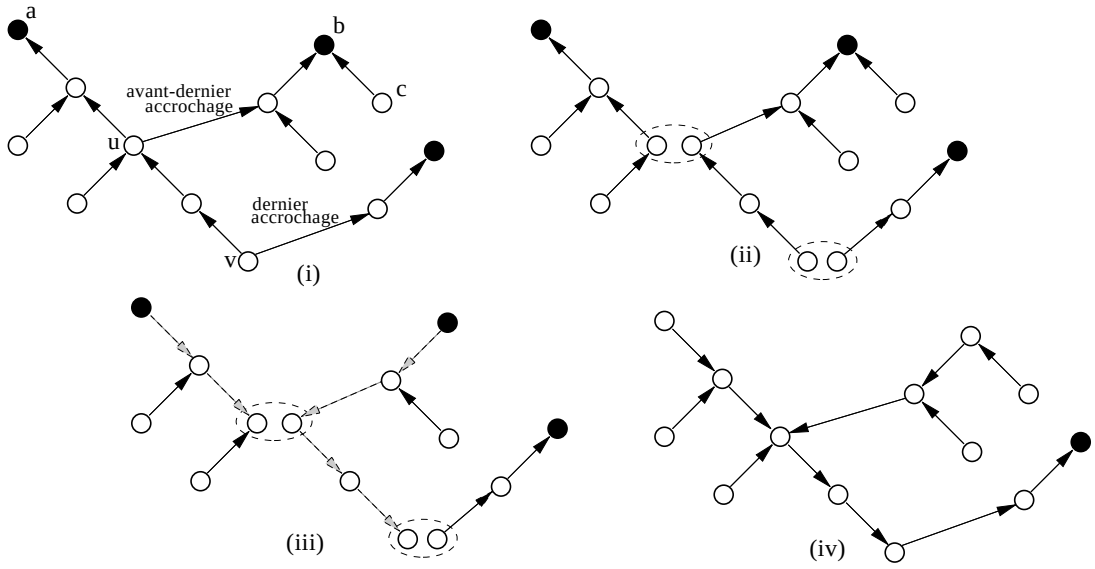


FIG. 1.3 – *(i) Orientation d'un arbre couvrant obtenue en orientant chaque arête réussissant à accrocher un arbre du sommet de l'étoile accrochée vers le sommet de l'arbre sur lequel elle est accrochée. (ii) Duplication des sommets qui ont plusieurs arcs sortants. (iii) Retournement des arêtes en direction du sommet de l'arbre de plus grande étiquette sauf dans l'arbre de racine c qui est celui sur lequel tous les autres se sont accrochés. (iv) Arbre couvrant orienté obtenu.*

qu'ils sont, au même titre que «*quicksort* », des algorithmes d'affinage de partition.

1.2 Structure de données compacte et algorithmes parallèles pour les graphes de permutation et les ordres de dimension fixée

Les graphes de permutation sont des objets combinatoires qui ont profité des progrès récents de l'algorithmique [3, 11, 37, 65, 76] qui sont liés à des techniques nouvelles de décomposition modulaire et d'orientation transitive (ces techniques seront abordées au chapitre 4). Par définition, les graphes de permutation possèdent un codage compact de taille n. En séquentiel, il est possible de passer du graphe à la permutation avec un travail $O(n+m)$ [58]. Le passage inverse se fait avec un travail $O(n^2)$ en séquentiel, et le problème est encore ouvert en parallèle. Nous verrons comment passer de la permutation au graphe dans le modèle CRCW PRAM avec un travail $O(m+n\log n)$. Nous en déduirons une nouvelle structure de données permettant de représenter le graphe en $O(n\log n)$.

De nombreux algorithmes nécessitent une représentation classique du graphe. Pour exécuter ces algorithmes sur un graphe de permutation, il est nécessaire d'en calculer une représentation de la liste des arêtes. Nous travaillerons dans le contexte orienté car les résultats se traduisent directement pour les graphes de permutation et il permet une généralisation. L'analogue orienté des graphes de permutation sont les ordres de dimension 2 qui se généralisent en les ordres de dimension d. Le problème de calculer efficacement la représentation classique d'un ordre de dimension d a été posé dans ces termes par SPINRAD [75]. Il est relié à celui de répondre efficacement à des requêtes géométriques dans un espace de dimension d, mais les méthodes utilisées dans ce contexte cachent la structure de données compacte que nous mettrons en évidence.

Après avoir donné les définitions nécessaires, nous commencerons par proposer un algorithme de calcul du nombre d'arcs d'un ordre de dimension 2 avec un travail $O(n \log n)$, ce qui est à un facteur $\log\log n$ du meilleur algorithme séquentiel [22, 32].

Nous mettrons ensuite en évidence une structure de données compacte provenant de l'exécution de cet algorithme, qui représente les ensembles de successeurs de chaque sommet en espace $O(n \log n)$. A l'inverse de la permutation, cette représentation permet l'utilisation de tous les algorithmes qui prennent les listes d'adjacence en entrée.

Nous verrons ensuite comment déduire les listes d'adjacence de cette structure de données. Nous calculerons aussi la réduction transitive de l'ordre de dimension 2 (la permutation représentant la fermeture).

Finalement, nous verrons comment généraliser ces résultats en dimension d quelconque.

Définitions

Une *permutation* π est une bijection de $\{0, \ldots, n-1\}$ dans lui-même, ou de manière équivalente un mot de n lettres avec toutes les lettres $0, \ldots, n-1$. Soit $\pi(i)$ l'image de i par π, ou encore la i^{e} lettre de π. π^{-1} désigne l'inverse de la bijection et $\widetilde{\pi}$ le mot renversé.

Les graphes et les ordres auront pour sommets $\{0, \ldots, n-1\}$. La dimension d'un ordre est le nombre minimal d d'ordres totaux sur ses sommets dont il est l'intersection (pour l'ensemble des arcs). Un ordre est toujours l'intersection de toutes ses extensions linéaires.Un jeu de d extensions linéaires dont l'ordre est l'intersection s'appelle un *réaliseur* (calculer un réaliseur est NP-complet pour les ordres de dimension supérieure à trois donné par une représentation classique).

Un ordre de dimension 2 est donné par deux ordres totaux, en numérotant ses sommets selon l'un des deux, le deuxième ordre total peut être donné par une permutation. Un graphe orienté $G = (V, \mathcal{A})$ est un *ordre de dimension 2* si et seulement si $\mathcal{A}$ est donné par une permutation π telle que $ij \in \mathcal{A}$ si et seulement

si $i < j$ et $\pi(i) < \pi(j)$ ou de manière équivalente si i apparaît avant j dans π^{-1}. Le *graphe de permutation* associé à π est le graphe de comparabilité de G.

Calcul du nombre d'arcs

Le nombre d'arcs de l'ordre de dimension 2 est le nombre d'inversions de $\widetilde{\pi^{-1}}$ puisqu'un arc ij avec $i < j$ n'est présent que lorsque i apparaît avant j dans le mot π^{-1}. Nous allons calculer ce nombre en triant $\widetilde{\pi^{-1}}$ à la manière de «*quicksort* ». A chaque phase de division du tri rapide, un compteur c est mis à jour de sorte que la somme de c et du nombre d'inversions de la permutation en cours de tri soit invariante.

Supposons sans perte de généralité $n = 2^q$. Les nombres triés étant $0, \ldots, n-1$, il est toujours facile de trouver un bon pivot qui divise exactement en deux les ensembles d'éléments considérés. L'algorithme fera donc toujours un nombre logarithmique de phases $\phi = 0, \ldots, q-1$.

Décrivons maintenant comment partager un bloc de taille $2^{q-\phi}$ de la permutation durant la phase ϕ en une suite de deux blocs en détectant certaines de ses inversions sans en créer de nouvelles.

Comme dans le tri rapide, les éléments plus grands (respectivement plus petits) que le pivot sont placés dans le bloc de droite (respectivement de gauche). L'ordre de la permutation précédente doit être préservé à l'intérieur de chaque bloc. En faisant cela, nous ôtons certaines inversions. Pour tout sommet i allant vers la gauche, il faut donc compter le nombre de sommets allant à droite qui apparaissaient à sa gauche et ajouter ce nombre au compteur c. Voir la figure 1.4.

Avant la phase initiale, on pose $W \leftarrow \widetilde{\pi^{-1}}$ et $c \leftarrow 0$. Durant la phase ϕ, les 2^ϕ blocs consécutifs de taille $2^{q-\phi}$ composant W sont coupés en deux. Par souci de clarté, l'algorithme est écrit pour le premier bloc (pour les autres, il faut simplement maintenir un compteur de position courante). Voir l'algorithme 1.2.

A la fin de la phase, le compteur c est mis à jour en lui ajoutant tous les $\Delta(v)$ grâce à une somme préfixée.

A la fin de l'algorithme, W est triée et n'a donc plus d'inversions, et c est alors le nombre d'inversions de $\widetilde{\pi^{-1}}$ et nous avons calculé le nombre $m = c$ d'arcs de l'ordre de dimension 2 associé à la permutation π. A la ligne 1, le pivot peut se calculer à partir du compteur de position courante qui se déduit facilement de la représentation binaire de x (voir le paragraphe qui suit sur la représentation compacte). Chaque phase est constituée essentiellement de deux sommes préfixées, ce qui requiert un temps $\log n$ avec $n/\log n$ processeurs. On en déduit le résultat suivant.

Théorème 1 *L'algorithme 1.2 permet de calculer le nombre d'arcs d'un ordre de dimension 2 donné par sa permutation en temps $O(\log^2 n)$ avec un travail $O(n \log n)$ dans le modèle* EREW PRAM.

Algorithme 1.2 Phase de division pour régner

Données : Un bloc $W(0), \ldots, W(2^{q-\phi}-1)$ d'une permutation.

Résultat : Le nombre $\Delta(W(x))$ de nouvelles inversions détectées pour chaque sommet $W(x)$ où $x \in \{0, \ldots, 2^{q-\phi}-1\}$.

Etape 1 *Comparaisons.*

1 Comparer chaque sommet au pivot $p = 2^{q-\phi-1}$.
Si $W(x) < p$ **Alors** $B(x) \leftarrow 0$ **Sinon** $B(x) \leftarrow 1$

Etape 2 *Inversions.*

Calculer les sommes préfixées $\sum_{i=0}^{x} B(i)$.
Si $B(x) = 0$ **Alors**
nous venons de détecter $\Delta(W(x)) \leftarrow \sum_{i=0}^{x} B(i)$ inversions relatives à $W(x)$
Sinon poser $\Delta(W(x)) \leftarrow 0$

Etape 3 *Diviser « à la quicksort ».*

Si $B(x) = 0$ **Alors**
Placer $W(x)$ en position $x - \sum_{i=0}^{x} B(i)$.
Sinon
Placer $W(x)$ en position $2^{q-\phi-1} + \sum_{i=0}^{x} B(i)$.

$\pi =$	2	4	7	0	5	6	1	3	
$\pi^{-1} =$	3	6	0	7	1	4	5	2	
$\widetilde{\pi^{-1}} =$	2	5	4	1	7	0	6	3	$c = 0$
$\phi = 1$	2 1		0 3		5 4		7 6		$c = 0 + 0 + 2 + 3 + 4 = 9$
$\phi = 2$	1 0		2 3		5 4		7 6		$c = 9 + 1 + 1 + 0 + 0 = 11$
$\phi = 3$	0	1	2	3	4	5	6	7	$c = 11 + 1 + 0 + 1 + 1 = 14$
									$m = c = 14$

FIG. 1.4 – *Une permutation et les différentes phases de l'algorithme. On a par exemple trouvé* 3 *inversions correspondant au sommet* 0 *à la phase 1 car* 5, 4 *et* 7 *apparaîssaient à sa gauche à la phase précédente.*

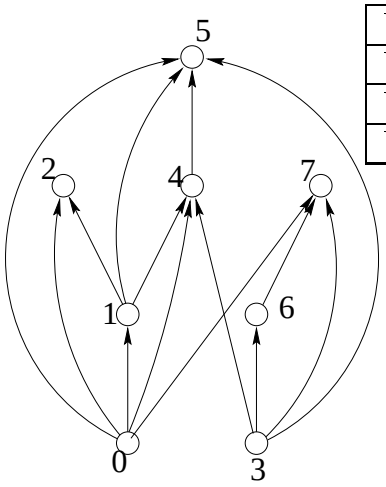
La complexité obtenue n'est pas très éloignée de celle du meilleur algorithme connu en séquentiel qui prend un temps $O\left(\frac{n \log n}{\log \log n}\right)$ [22, 32].

Une représentation compacte des listes d'adjacence

En fait, l'algorithme précédent calcule implicitement une représentation particulière des adjacences de l'ordre de dimension 2 que nous allons maintenant expliciter. Gardons pour cela des copies W_ϕ et Δ_ϕ des vecteurs obtenus à chaque

phase (voir la figure 1.5). (W_ϕ est une copie de W avant la phase ϕ.)

Considérons l'algorithme sur l'ordre de dimension 2. Le nombre $\Delta_\phi(v)$ de nouvelles inversions relatives à v correspondent à $\Delta_\phi(v)$ successeurs de v. Si v va dans un bloc de droite, aucun de ses successeurs n'est détecté durant la phase. Dans le cas contraire, v est plus petit que le pivot, alors que les sommets allant à droite sont plus grands. Ceux qui apparaissaient de plus à sa gauche font par conséquent partie de ses successeurs. Ce sont exactement les $\Delta_\phi(v)$ premiers éléments du bloc de droite et ils forment un intervalle $I_\phi(v) = [g_\phi(v), d_\phi(v)]$ de W_ϕ. Remarquons que les autres successeurs de v vont dans le bloc de gauche avec v et seront détectés plus tard.



	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
W_0	2	5	4	1	7	0	6	3
W_1	2	1	0	3	$]_2$5	4$]_1$	7$]_0$	6$]_3$
W_2	1	0	$[2]_1]_0$	3	5	4 $[]_5]_4$7		6
W_3	0	$[1]_0$	2	$[]_2$3	4	$[5]_4$	6	$[7]_6$

	Degré$_1$	*Degré*$_2$	*Degré*$_3$	I_1	I_2	I_3
0	3	1	1	$[4, 6]$	$[2, 2]$	$[1, 1]$
1	2	1	0	$[4, 5]$	$[2, 2]$	$\emptyset$
2	0	0	0	$\emptyset$	$\emptyset$	$\emptyset$
3	4	0	0	$[4, 7]$	$\emptyset$	$\emptyset$
4	0	0	1	$\emptyset$	$\emptyset$	$[5, 5]$
5	0	0	0	$\emptyset$	$\emptyset$	$\emptyset$
6	0	0	1	$\emptyset$	$\emptyset$	$[7, 7]$
7	0	0	0	$\emptyset$	$\emptyset$	$\emptyset$

FIG. 1.5 – *L'ordre de dimension 2 de la figure 1.5 et les intervalles de sa représentation compacte.*

$I_\phi(v)$ peut être calculé durant l'étape ϕ comme suit. $g_\phi(v)$ est le compteur de position courante calculé pour le bloc de droite à l'étape $\phi + 1$ et on a $d_\phi(v) = g_\phi(v) + \Delta_\phi(v)$. Soit x l'index de $v = W(x)$ et $x = b_1 \cdots b_q$ sa représentation binaire (b_1 est le bit de poids fort). On a alors $g_\phi(x) = \underbrace{b_1 \ldots b_\phi 10 \ldots 0}_{q}$.

Nous pouvons formaliser ce concept de représentation par des intervalles d'ordres totaux sur V à l'aide de la définition suivante.

Définition 2 *Soit $G = (V, \mathcal{A})$ un graphe orienté, et pour un entier k fixé, soient $W_1, \ldots, W_k$ des tableaux représentant des sous-ensembles de V totalement ordonnés. Pour chaque sommet v, soient $I_1(v), \ldots, I_k(v)$ des intervalles de $W_1, \ldots, W_k$ respectivement. $I_\phi(v) = [l_\phi(v), r_\phi(v)]$ est l'ensemble des éléments de W_ϕ d'indice compris entre $l_\phi(v)$ et $r_\phi(v)$.*

Nous dirons que $W_1, \ldots, W_k$ *et* $I_1, \ldots, I_k$ *forment une* k-représentation intervallaire compacte *de* G *quand l'ensemble des successeurs de chaque sommet* $v \in V$ *est* $\bigcup_{i=1}^{k} I_i(v)$.

Un ordre total donné par un tableau W contenant ses éléments triés selon cet ordre admet une 1-représentation intervallaire compacte triviale où $W_1 = W$ et $I_1(W(x)) = [x+1, n-1]$ pour tout sommet $W(x)$. Tout graphe de n sommets a une n-représentation intervallaire compacte triviale (isomorphe à l'ensemble des listes d'adjacence du graphe) où W_v est la liste des successeurs de v et $I_v(v) = [1, |W_v|]$ et $I_\phi(v) = \emptyset$ si $\phi \neq v$.

Nous avons vu comment calculer en temps $O(\log^2 n)$ avec un travail $O(n \log n)$ une $\log n$-représentation intervallaire compacte (voir la figure 1.5).

Dans le cas des tableaux calculés par l'algorithme 1.2, chaque W_ϕ et les intervalles associés représentent en fait ce qui s'appelle un ordre de *contiguïté* (u précède v dans cet ordre si et seulement si $v \in I_\phi(u)$) car W_ϕ (en tant que permutation des sommets représentant un ordre total) forme une extension linéaire de cet ordre. Nous avons donné une représentation générale pour la représentation compacte de sorte qu'elle pourrait permettre de représenter aussi des graphes quelconques.

De ce point de vue, cette définition est à rapprocher de travaux de Christian Capelle [8] qui a étudié comment représenter un ordre quelconque comme une union d'ordres d'intervalles (qui admettent eux aussi une représentation linéaire en leur nombre de sommets). Il propose un algorithme qui calcule une telle représentation pour un ordre quelconque (cette représentation nécessite $O(n)$ ordres intervalles dans le pire cas et une optimisation permettrait, semble-t-il, d'obtenir une représentation en $O(n\sqrt{n})$). D'autre part, cette représentation est plutôt une représentation compacte de la matrice d'adjacence (elle permet de tester si deux éléments sont comparables), alors que la représentation intervallaire compacte proposée ici est plus proche des listes d'adjacence des sommets :

Théorème 3 *Tout algorithme de traitement de graphes utilisant une représentation sous forme de listes d'adjacence de l'entrée peut utiliser à la place une* k*-représentation intervallaire compacte.* m *devient alors* $kn + m$ *dans la complexité de l'algorithme.*

Une k-représentation intervallaire compacte représente la liste d'adjacence de chaque sommet par k intervalles, l'inspection de la liste d'adjacence d'un sommet u demande donc un travail $O(k + \mathit{Degré}(u))$ au lieu de $O(\mathit{Degré}(u))$.

Nous verrons un peu plus loin, en généralisant aux ordres de dimension d le calcul d'une représentation intervallaire compacte, que la représentation calculée par la généralisation de l'algorithme 1.2 fournit de plus un recouvrement de l'ordre d'entrée par une union d'ordres d'intervalles de hauteur 1. Remarquons que cet algorithme prend en entrée une représentation de l'ordre sous forme d'intersection d'extensions linéaires, on ne sait pas calculer dans le cas général une

telle représentation minimale (avec un nombre minimal d'extensions linéaires). Il serait intéressant d'étudier de plus près les relations entre ces deux représentations d'ordres : par une union d'ordres d'intervalles et par une union d'ordres de contiguïté.

Calcul d'une représentation classique à partir d'une représentation compacte

Nous allons maintenant voir comment calculer une représentation classique, c'est-à-dire la liste explicite des arcs, à partir d'une représentation compacte. La taille d'une représentation sous forme de listes d'adjacence étant m, ce nombre doit être d'abord calculé puisque c'est aussi le nombre de processeurs qui seront utilisés.

Algorithme 1.3 Calcul d'une représentation classique

Données : k tableaux $W_1, \ldots, W_k$ de taille n et un tableau de taille nk d'intervalles $I_\phi(v) = [g_\phi(v), d_\phi(v)]$, pour $0 \leq v < n$ et $1 \leq \phi \leq k$.

Résultat : Les listes d'adjacence.

Début

Calculer le nombre m d'arcs.
Allouer un tableau A de taille m. {*Chaque élément de A contiendra un arc.*}
Calculer l'origine de chaque arc.
Calculer la destination de chaque arc.

Fin

Rappelons que dans les algorithmes détaillés qui suivent, l'ensemble des sommets de tout graphe est $\{0, \ldots, n-1\}$.

Algorithme 1.4 Calcul du nombre d'arcs

Début

Pour tout *intervalle* $I_\phi(v)$ **effectuer** $Degré_\phi(v) \leftarrow$ longueur de $I_\phi(v)$
Allouer un tableau D de taille nk.
Pour tout $1 \leq \phi \leq k$ *et* $0 \leq v \leq n-1$ **effectuer** $D(kv+\phi) \leftarrow Degré_\phi(v)$
1 Calculer les sommes préfixées $S_\phi(v) := \sum_{a=0}^{kv+\phi-1} D(a)$.
$m \leftarrow S_k(n-1)$.

Fin

On peut remarquer que les valeurs $S_\phi(v)$ calculées à la ligne 1 sont égales à $\sum_{u=0}^{v-1} Degré^+(u) + \sum_{\psi=1}^{\phi-1} Degré_\psi(v)$.

Le degré d'un sommet étant $S_{k+1}(v) - S_1(v)$, A.origine ressemble à ceci :

$$A.\text{origine} = \underbrace{0 \cdots 0}_{Degré^+(0)} \underbrace{1 \cdots 1}_{Degré^+(1)} \cdots\cdots \underbrace{n-1 \cdots n-1}_{Degré^+(n-1)}$$

Algorithme 1.5 Calcul des origines des arcs

Résultat : Le tableau trié A.origine des origines des arcs.
Début
Initialiser un tableau T de taille m à 0.
Pour tout $0 \leq v < n$ **effectuer** $T(S_1(v)) \leftarrow 1$
Calculer les sommes préfixées $A.\text{origine}(a) := \sum_{b=0}^{a} T(b)$.
Fin

Algorithme 1.6 Calcul des destinations des arcs

Résultat : Le tableau A.destination des destinations des arcs.
Début
Initialiser un tableau A.phase de taille m à O.
Pour tout $0 \leq v < n$ *et* $1 \leq \phi \leq k$ **effectuer** $A.\text{phase}(S_\phi(v)) \leftarrow 1$
Calculer les sommes préfixées $A.\text{phase}(a) \leftarrow \bigoplus_{b=0}^{a} A.\text{phase}(b)$.
Pour tout $0 \leq a < m$ **effectuer**
$v := A.\text{destination}(a)$
$\phi := A.\text{phase}(a)$
$A.\text{destination}(a) := W_\phi\left(g_\phi(v) + a - S_\phi(v)\right)$
Fin

L'opération $\oplus$ est utilisée pour calculer la somme préfixée dans chaque bloc correspondant à la liste d'adjacence d'un sommet. Avec $A(a) = \big(A.\text{origine}(a), A.\text{phase}(a)\big) = (u, \phi)$ et $A(b) = (v, \psi)$, elle est définie par $A(a) \oplus A(b) = (v, \psi)$ quand $u < v$ et $A(a) \oplus A(b) = (u, \phi + \psi)$ quand $u = v$. Après le calcul des sommes préfixées, A.phase ressemble donc à :

$$A.\text{phase} = \underbrace{\underbrace{1\cdots1}_{\Delta_1(0)}\cdots\underbrace{k\cdots k}_{\Delta_k(0)}}_{Degré^+(0)} \quad \underbrace{\underbrace{1\cdots1}_{\Delta_1(1)}\cdots\underbrace{k\cdots k}_{\Delta_k(1)}}_{Degré^+(1)} \quad \cdots\cdots \quad \underbrace{\underbrace{1\cdots1}_{\Delta_1(n-1)}\cdots\underbrace{k\cdots k}_{\Delta_k(n-1)}}_{Degré^+(n-1)}$$

La dernière instruction requiert une lecture concurrente. Etant donnée la complexité des sommes préfixées, on obtient :

Théorème 4 *Soit G un graphe orienté donné par une k-représentation intervallaire compacte. L'algorithme 1.3 calcule la représentation classique sous forme de listes d'adjacence du graphe en temps $O(\log n)$ avec un travail $O(m + nk)$ dans le modèle* CREW PRAM.

En combinant ce résultat avec l'algorithme de calcul de la représentation compacte, on obtient :

Théorème 5 *La combinaison des algorithmes 1.2 et 1.3 permet de calculer la représentation classique sous forme de listes d'adjacence d'un ordre de dimension 2*

donné par sa permutation en temps $O(\log^2 n)$ *avec un travail* $O(m+n\log n)$ *dans le modèle* CREW PRAM.

Calcul de la réduction transitive

L'algorithme précédent calcule les arcs de la fermeture transitive de l'ordre de dimension 2. Nous allons voir maintenant comment calculer ceux de la réduction transitive (voir la figure 1.6).

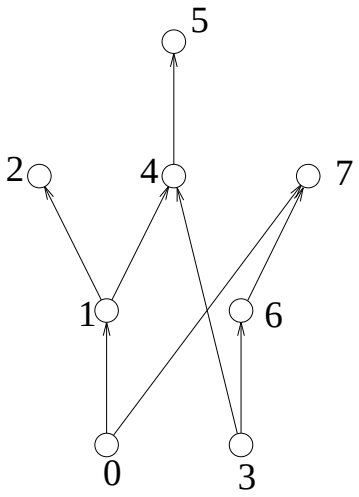


FIG. 1.6 – *La réduction transitive de l'ordre de la figure 1.5.*

Un arc ij de la fermeture transitive est un arc de la réduction transitive si de plus il n'existe pas de k tel que $i < k < j$ et $\pi(i) < \pi(k) < \pi(j)$, ou de manière équivalente s'il n'existe pas de k tel que $i < k < j$ apparaissant entre i et j dans π^{-1}.

Soit S_i la liste des successeurs de i dans l'ordre où ils apparaissent dans π^{-1}. Les arcs ij de la réduction transitive sont ceux qui vérifient $j = S_i(p)$ et $j \geq S_i(q)$ $\forall q < p$, ou encore :

$$j \in S_i \ (j = S_i(p)) \text{ et } j = \min S_i(1), S_i(2), \ldots, S_i(p) \ .$$

On peut faire ce calcul de minima avec une somme préfixée si les listes d'adjacence sont triées en accord avec π^{-1}. Un simple test d'égalité permet ensuite de déterminer si un arc fait partie de la réduction transitive. En comptant le tri des listes d'adjacence, on obtient :

Théorème 6 *On peut calculer la réduction transitive d'un ordre de dimension 2 donné par sa permutation en temps* $O(\log^2 n)$ *avec un travail* $O((n+m_t)\log n)$ *dans le modèle* CREW PRAM *où* m_t *est le nombre d'arcs de la fermeture transitive de l'ordre.*

Représentation compacte des ordres de dimension fixée

Un ordre P est de *dimension* d s'il admet un *réaliseur* qui peut être représenté par d permutations $\pi_1, \ldots, \pi_d$. $i <_P j$ pour cet ordre si et seulement si $\pi_t(i) < \pi_t(j)$ pour tout t entre 1 et d. Dans ce qui précède, on avait implicitement choisi l'identité pour π_1 et π pour π_2. Chaque π_t induit une extension linéaire de l'ordre puisque $i <_P j$ implique $\pi_t(i) < \pi_t(j)$.

Cette notion a des applications en base de données. La question « quels sont les successeurs de i? » pourrait être la traduction d'une requête du type « quels sont les membres de la base de données de plus de trente ans, de sexe mâle, ayant les cheveux châtains ou noirs, et ayant publié au moins 100 articles ? » Nous reviendrons sur ce problème un peu plus loin.

Nous allons maintenant reprendre l'algorithme proposé pour les ordres de dimension sous un autre angle pour pouvoir le généraliser plus facilement. L'idée est de représenter l'ordre avec une permutation de moins en remplaçant implicitement l'une d'elles par l'ordre des numéros des sommets n'est utile que dans le cas des ordres de dimension 2, pour faciliter la compréhension, il vaut mieux s'en débarrasser. Considérons que les sommets sont maintenant des couples $v(i) = (\pi_1(i), \pi_2(i))$.

L'idée de l'algorithme que nous avons proposé est de couper en deux l'ensemble des sommets selon la première coordonnée : $G = \{(x, y) | 0 \leq x < n/2\}$ et $D = \{(x, y) | n/2 \leq x < n\}$. Puis on résout récursivement le problème sur G et sur D. La partie difficile est de trouver les arcs de G vers D. Pour cela l'idée de base est de ne garder que les arcs uv avec $u \in G$ et $v \in D$ dans l'ordre de dimension 1 $(\pi_2(0)), \ldots, (\pi_2(n-1))$. Dans cet ordre total, les successeurs de $u = \pi(i)$ sont les $v = \pi(j)$ tels que $\pi(i) < \pi(j)$.

Pour ne pas avoir à faire un tri supplémentaire, l'algorithme précédent commençait implicitement par trier les sommets une fois pour toutes selon la deuxième coordonnée, c'est-à-dire dans l'ordre :

$$(\pi_1(\pi_2^{-1}(n-1)), n-1), \ldots, (\pi_1(\pi_2^{-1}(1)), 1), (\pi_1(\pi_2^{-1}(0)), 0).$$

Cela permettait d'identifier les successeurs dans D de $u \in G$ comme un intervalle de D. Cette technique permet de gagner un facteur $\log n$ en travail.

L'idée naturelle pour généraliser ce résultat (et c'est celle que l'on trouve dans la littérature [66]) consiste à résoudre récursivement trois problèmes plus petits : deux problèmes sur $n/2$ sommets en dimension d et un problème sur n sommets en dimension $d-1$ (on manie des d-uplets). Couper l'ensemble V des n d-uplets en deux parties égales G et D selon leur première coordonnée : G (respectivement D) est l'ensemble des sommets de première coordonnée plus petite (respectivement plus grande) que le pivot. Résoudre récursivement les deux problèmes de dimension d et de taille $n/2$ sur G et sur D et le problème de dimension $d-1$ sur les n $d-1$-uplets obtenus en effaçant la première coordonnée. Tout arc de l'ordre de dimension d est soit un arc de l'ordre induit sur G, soit un

arc de l'ordre induit sur D, soit un arc de G vers D dans l'ordre de dimension $d-1$ sur V où l'on oublie la première coordonnée (par définition, la première coordonnée d'un d-uplet de G est toujours plus petite que celle d'un d-uplet de D). Cet algorithme permet de calculer une $\log^d n$ représentation intervallaire compacte en temps $O(\log^{d+1} n)$ avec $dn/\log n$ processeurs, soit un travail en $O(dn \log^d n)$.

Toute explication plus détaillée de cet algorithme est difficile à comprendre. La structure de données compacte introduite un peu plus tôt va nous permettre de nous éviter cette peine en nous permettant de donner un algorithme non récursif basé sur la procédure de partitionnement de l'algorithme 1.2.

Nous allons voir comment calculer une représentation intervallaire compacte de l'intersection d'un ordre $P = (V, <)$ donné par une représentation intervallaire compacte et un ordre total $T = (V, <_{tot})$ sur V donné par sa 1-représentation intervallaire compacte qu'est la permutation π associée.

Considérons sous cet angle l'algorithme 1.2 : on part de la 1-représentation associée à l'une des permutations et en la partitionnant $\log n$ fois, on obtient en gardant des copies des tableaux intermédiaires une $\log n$-représentation intervallaire compacte de l'intersection des deux ordres totaux associés aux deux permutations. Cette idée se généralise facilement : nous allons voir comment calculer une $k \log n$-représentation intervallaire compacte de $P \cap T$ en temps $O(\log^2 n)$ avec un travail de $O(n \log n)$, ce qui au total nous fera encore économiser un facteur $\log n$.

Algorithme 1.7 Calcul d'une représentation compacte

Données : d permutations $\pi_1, \ldots, \pi_d$ de $\{0, \ldots, n-1\}$.

Résultat : Un ensemble $\mathcal{W}$ de $\log^{d-1} n$ tableaux et les intervalles de successeurs $I_f(x) = [g_f(x), d_f(x)]$ associés à chaque sommet $W_f(x)$.

Début

Trier $\{0, \ldots, n-1\}$ selon l'ordre π_1 pour obtenir le tableau W_1 ($x <_{\pi_1} y$ si et seulement si $\pi_1(x) < \pi_1(y)$).

Pour tout $0 \leq x < n$ **effectuer** $I_1(x) \leftarrow [x+1, n-1]$

Poser $\mathcal{W} = \{W_1\}$.

Pour $\delta = 2$ *à* d **effectuer**

Pour tout $W_f \in \mathcal{W}$ **effectuer**

Calculer une représentation compacte $W_{f,0}, \ldots, W_{f,\log n}$, $I_{f,0}, \ldots, I_{f,\log n}$ de l'intersection de l'ordre de contiguïté représenté par W_f et I_f avec l'ordre total induit par π_δ en triant W_f « à la quicksort » selon $<_{\pi_\delta}$ (voir algorithme 1.8).

Poser $\mathcal{W} \leftarrow \bigcup_{W_f \in \mathcal{W}} \{W_{f,0}, \ldots, W_{f,\log n}\}$.

Fin

Chacun des tableaux W_f, $1 \leq f \leq k$, de la k-représentation de P et les in-

tervalles associés représentent un ordre P_f (de contiguïté). P est l'union de ces ordres. La distributivité de l'intersection par rapport à l'union nous dit que $P \cap T$ est l'union des $P_f \cap T$. Il suffit donc d'appliquer l'algorithme des ordres de dimension 2 à chacun des tableaux W_f avec une légère modification de l'algorithme 1.2, tout le reste en découle facilement. L'algorithme 1.7 donne la trame générale et l'algorithme 1.8 est une modification de l'algorithme 1.2 qui permet de calculer une $\log n$-représentation intervallaire compacte associée à l'intersection d'un ordre total avec le graphe orienté représenté par un tableau W_f et des intervalles $I_f(x)$ (voir aussi la figure 1.7).

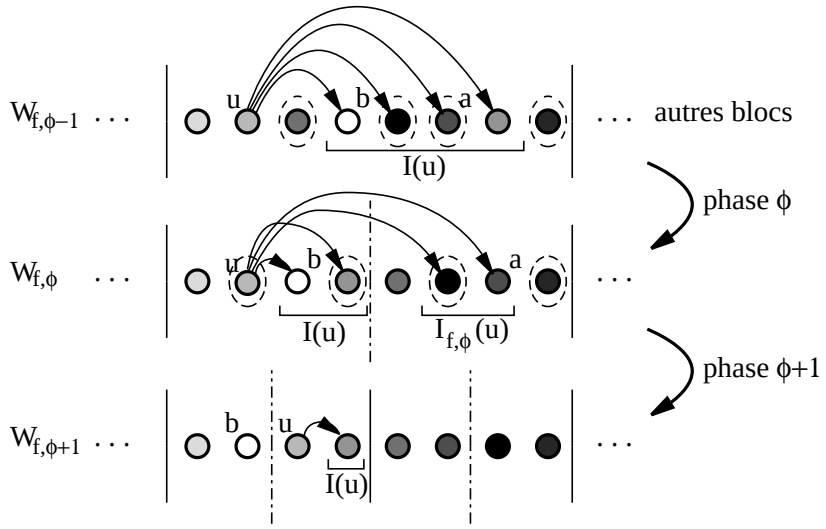


FIG. 1.7 – *Affinage de l'intervalle des successeurs d'un sommet u lors d'une phase de division. a est identifié dans la première phase de division comme un successeur de u après intersection avec l'ordre total donné par les couleurs des sommets, et b est écarté (n'étant pas un successeur de u après intersection avec l'ordre total) à la deuxième phase de division.*

Théorème 7 *L'algorithme 1.7 permet de calculer une* $\log^{d-1} n$*-représentation intervallaire compacte d'un ordre de dimension d donné par un réaliseur.*

L'algorithme 1.8 peut être généralisé : si deux jeux d'intervalles pour un même tableau W_f représentent deux graphes orientés, cet algorithme permet de calculer les $\log n$ tableaux et les intervalles associés à deux représentations compactes de l'intersection de chacun des graphes de départ avec T. Il suffit pour cela d'effectuer les mêmes opérations sur le deuxième jeu d'intervalles que sur le premier. Le problème étant symétrique, on peut aussi calculer pour les mêmes tableaux les intervalles associés avec l'intersection avec l'ordre inverse de T (pour lequel $u <$

Algorithme 1.8 Intersection avec un ordre total

Données : Un tableau W_f et les intervalles $I_f(x) = [g(x), d(v)]$ associés représentants un graphe orienté. Un ordre total T induit par une permutation π.

Résultat : Un bloc $W_{f,\phi}(0), \ldots, W_{f,\phi}(2^{q-\phi}-1)$ de $W_{f,\phi}$ et les intervalles de successeurs $I_{f,\phi}(x) = [g_{f,\phi}(x), d_{f,\phi}(x)]$ associés aux sommets du bloc.

Début

Pour tout $0 \leq x < n$ **effectuer**

$I(x) \leftarrow I_f(x)$

$W_{f,0}(x) \leftarrow W_f(x)$

Pour $\phi = 0$ *à* $\log n - 1$ **effectuer**

Etape 1 *Comparaisons.*

Comparer chaque sommet au pivot $p = g_\phi(x)$ de son bloc.{*Son bloc est l'ensemble des éléments d'indice compris entre* $g_{\phi-1}(x)$ *et* $g_{\phi-1}(x) + 2^{q-\phi} - 1$.}

Si $\pi(W_{f,\phi}(x)) < \pi(p)$ **Alors** $B(x) \leftarrow 0$ **Sinon** $B(x) \leftarrow 1$ {*Chaque bloc de longueur* $2^{q-\phi}$ *est coupé en deux: si* $B(x) = 0$, *l'élément* $\pi(W_{f,\phi}(x))$ *va dans le sous-bloc de gauche, et dans le sous-bloc de droite sinon.* }

Etape 2 *Diviser « à la quicksort ».*

Calculer les sommes préfixées $S_{f,\phi}(x) = \sum_{i=0}^{x-1} B(i)$.

Si $B(x) = 0$ **Alors**

Placer $W_{f,\phi}(x)$ en position $G_{f,\phi}(x) = x - S_{f,\phi}(x))$.

Sinon

Placer $W_{f,\phi}(x)$ en position $D_{f,\phi}(x) = 2^{q-\phi-1} + S_{f,\phi}(x)$.

Plus précisément :

Pour tout $0 \leq x < 2^{q-\phi}$ **effectuer**

Si $B(x) = 0$ **Alors** $P(x) \leftarrow G_{f,\phi}(x)$ **Sinon** $P(x) \leftarrow D_{f,\phi}(x)$

$W_{f,\phi+1}(P(x)) \leftarrow W_{f,\phi}(x)$

Si $B(x) = 0$ **Alors** $I(x) \leftarrow [G_{f,\phi}(g(x)), G_{f,\phi}(d(x))]$

Sinon $I(x) \leftarrow [D_{f,\phi}(g(x)), D_{f,\phi}(d(x))]$

Etape 3 *Successeurs identifiés.*

1 **Si** $B(x) = 0$ **Alors** $I_{f,\phi}(x) \leftarrow [D_{f,\phi}(g(x)), D_{f,\phi}(d(x))]$

Sinon $I_{f,\phi}(x) \leftarrow \emptyset$

Fin

v si et seulement si $v <_T u$). Il suffit pour cela de remplacer la ligne 1 par :

Si $B(x) = 0$ **Alors** $I_{f,\phi}(x) \leftarrow \emptyset$ **Sinon** $I_{f,\phi}(x) \leftarrow [G_{f,\phi}(g(x)), G_{f,\phi}(d(x))]$

On peut donc aussi représenter les prédécesseurs dans les mêmes tableaux avec le même nombre d'intervalles : en utilisant les intervalles que l'on a posés égaux à l'ensemble vide et qui correspondent en fait à la détection de prédécesseurs (le problème est symétrique). Cela est utile dans l'application aux bases de données où une requête sera plutôt du type « Quels sont les membres de la liste de sexe féminin, ayant entre 20 et 30 ans, ayant entre 1 et 2 enfants,... ». La $\log^{d-1} n$-représentation intervallaire compacte permet de répondre à une telle requête en temps séquentiel $O(\log^d n)$. La question se traduit par « Quels sont les successeurs de u qui sont aussi des prédécesseurs de v ? » où u et v sont deux nouveaux sommets.

Voici comment trouver les successeurs d'un nouveau sommet u. On calcule en temps $O(\log n)$ la position où u s'insère dans la liste triée selon la première coordonnée. On simule l'algorithme de calcul de la représentation compacte en suivant le cheminement de u dans les blocs de tableaux correspondants aux résultats des comparaisons de u avec les pivots (rappelons que les pivots se calculent à partir de la représentation binaire de u). Pour cela, il suffit de conserver une copie des tableaux $G_{f,\phi}$ et $D_{f,\phi}$ (qui se déduisent des sommes préfixées $S_{f,\phi}$) au moment où on a calculé la représentation intervallaire compacte (cela n'est pas nécessaire on peut trouver la position dans le tableau suivant en regardant l'intervalle associé à un voisin de u et éventuellement l'intervalle associé au sommet correspondant à une borne de cette intervalle). Cela nécessite un temps $O(\log^{d-1} n)$ car on calcule juste les intervalles. u n'est pas inséré dans la représentation. Voir l'algorithme 1.9.

Dans l'application de la base de données ou dans le contexte géométrique de points dans un espace de dimension d, $(y_1, \ldots, y_d)$ sont des réels et on n'a pas d'écriture binaire pour y_δ. Il faut remplacer b_ϕ par le résultat de la comparaison de y_δ avec le pivot du bloc où se trouve le sommet.

Théorème 8 *L'algorithme 1.9 permet de calculer les intervalles de successeurs d'un nouveau sommet en temps $O(d \log n)$ avec un travail $O(d \log^{d-1} n)$ dans le modèle* CREW PRAM. *En séquentiel, il s'exécute en temps $O(\log^{d-1} n)$.*

La représentation permettant dans le cas des ordres de dimension d de représenter aussi les ensembles de prédecesseurs, on peut calculer de manière analogue les intervalles correspondant aux prédecesseurs de v. Les intervalles représentant les sommets à la fois successeurs de u et prédecesseurs de v sont obtenus en combinant les deux intervalles obtenus pour u et v à chaque tableau $W_{f,\phi}$. Si v n'est pas un successeur de u, tous les intervalles sont vides. Sinon, chaque fois que l'on divise un bloc par un pivot p, soit $u < p < v$, soit u et v vont dans le même sous-bloc. Dans le premier cas, on combine $I_{f,\phi}(u)$ et $I_{f,\phi}(v)$ par union et par intersection dans le deuxième cas (on obtient bien un intervalle dans les deux cas). Finalement, on obtient des intervalles représentant la réponse à la requête en temps séquentiel $O(\log^{d-1} n)$. Pour obtenir la liste explicite des sommets de

Algorithme 1.9 Calcul des intervalles d'un nouvel élément

Données : Un élément $u = (y_1, \ldots, y_d)$ et une représentation intervallaire compacte doonnée par des tableaux W_f et les sommes préfixées associées.

Résultat : Les intervalles $I_f(u)$ de successeurs de u.

Début

$y_1, \ldots, y_d$ sont considérés comme les valeurs de u par $\pi_1, \ldots, \pi_d$.
u s'insère donc en position y_1 dans W_1.
Poser $I_1(u) \leftarrow [y_1 + 1, n - 1]$ et $E_{\mathcal{W}} \leftarrow \{1\}$.
Pour $\delta = 2$ *à* d **effectuer**
  Soit $b_1 b_2 \cdots b_{q-1} = y_\delta$ la représentation binaire de y_δ.
  Pour tout $f \in E_{\mathcal{W}}$ **effectuer**
    Soient a et b les bornes de $I_f(u) = [a, b]$.
    Pour $\phi \leftarrow 1$ *à* $\log n$ **effectuer**
      Si $b_\phi = 0$ **Alors**
        $I_{f,\phi}(u) \leftarrow [D_{f,\phi}(a), D_{f,\phi}(b)]$
        $a \leftarrow G_{f,\phi}(a)$
        $b \leftarrow G_{f,\phi}(b)$
      Sinon
        $I_{f,\phi}(u) \leftarrow \emptyset$
        $a \leftarrow D_{f,\phi}(a)$
        $b \leftarrow D_{f,\phi}(b)$
  Poser $E_{\mathcal{W}} \leftarrow \bigcup_{f \in E_{\mathcal{W}}} \{(f, 1), \ldots, (f, \log n)\}$.

Fin

la réponse, cela demande un travail suplémentaire de l'ordre du nombre de tels sommets [4].

Remarque. Cet algorithme est un algorithme d'affinage de partition (voir chapitre 4). En effet, les W_f sont obtenus en affinant une partition de l'ensemble des sommets. Les blocs sont ce que l'on appellera boîtes au chapitre 4. On commence par couper W_f selon un pivot $v(x)$ (tel que $\pi(x) = n/2$) en deux blocs ou boîtes $W_f \cap Succ_{<_{tot}}(x)$ et $W_f \cap \overline{Succ_{<_{tot}}(x)}$. Puis on coupe chaque bloc avec un pivot qui lui est propre.

Remarquons que l'algorithme optimal séquentiel pour les ordres de dimension 2 consiste à prendre les pivots dans l'ordre $<_{tot}$. Comme $Succ_{<_{tot}}(\pi^{-1}(0)) = V - \{v(\pi(0))\}$, on peut faire la partition en temps constant en isolant $v(\pi(0))$ du

4. Le problème est analogue à celui de trouver les δ successeurs d'un sommet à partir d'une représentation compacte, ce qui se fait en exécutant l'algorithme 1.3 pour un seul sommet en temps $O(\log(\log^{d-1} n))$ avec un travail $O(\delta + \log^{d-1} n)$

reste des sommets et identifier les successeurs de $v(\pi(0))$ en temps $Degré^+(v(\pi(0)))$. On obtient par conséquent un algorithme séquentiel en $O(n+m)$ pour calculer la représentation classique de l'ordre de dimension 2.

Sous cet angle, le résultat se généralise très simplement aux ordres de dimension d en partant d'une représentation intervallaire compacte de l'intersection de $d-1$ extensions linéaires pour donner un algorithme séquentiel en $O(n \log^{d-2} n + m)$.

En parallèle, on est obligé de couper en classes de tailles comparables pour obtenir un temps de calcul polylogarithmique et un travail non quadratique. Remarquons enfin que tous les tableaux d'une représentation intervallaire compacte sont des tableaux de numéros de sommets (des permutations même) et non des d-uplets. Il n'y a donc pas de constante d cachée dans les grands O.

Nous allons maintenant voir comment la représentation compacte calculée par l'algorithme 1.7 fournit aussi une représentation sous forme d'union d'ordres d'intervalles selon [8]. Examinons enfin d'un point de vu théorique la forme de la représentation compacte calculée par l'algorithme 1.7. Considérons tout d'abord l'ordre P induit par un tableau W_f de la représentation. Les seules relations d'ordre qu'il contient relient un sommet d'un sous-bloc de gauche à un sommet d'un sous-bloc de droite. Un sommet d'un sous-bloc de droite n'a donc pas de successeur dans cet ordre qui est donc biparti, c'est-à-direde hauteur 1 (la hauteur d'un ordre est la longueur d'un plus long chemin reliant un minimum à un sommet). D'autre part, l'ordre total induit par la position des sommets dans W_f est une extension linéaire de P puisque la destination d'un arc de P apparaît toujours à droite de son origine dans W_f. W_f est donc une extension linéaire de P telle que les successeurs de tout sommet sont consécutifs dans cette extension, ce qui prouve que P est un ordre de contiguïté.

Considérons maintenant l'algorithme 1.8 comme un algorithme de partitionnement (voir la figure 1.7) où l'intervalle de successeurs potentiels de chaque sommet est coupé en deux par l'affinage de partition effectué lors d'une phase de division. Un ordre d'intervalle est caractérisé par la propriété suivante : les ensembles de successeurs sont totalement ordonnés par inclusion. Si un intervalle I est inclus dans un intervalle I', alors $I \cap D \subseteq I' \cap D$ où D est le sous-bloc de droite. L'ordre total de départ est un ordre d'intervalle : l'intervalle des successeurs d'un sommet contient l'intervalle de successeurs de tous sommet à sa droite dans W_0. Cette propriété se conserve donc à l'intérieur de chaque bloc, ce qui permet d'affirmer qu'un bloc d'un tableau W_ϕ et les intervalles de successeurs associés représente toujours un ordre d'intervalle (qui est de plus biparti et de contiguïté). La représentation compacte d'un ordre de dimension d calculée par l'algorithme 1.7 fournit donc des ordres d'intervalles dont il est l'union. Ces ordres sont en nombre $O(n^{d-1})$, ce qui est loin des $O(n^2)$ de [8] mais la taille de la représentation est en $O(n \log^{d-1} n)$ (dans le pire cas, on a $d = n/2$).

Il serait intéressant d'étudier le nombre minimal d'ordres de contiguïté dont

l'union est un ordre donné. Existe-t-il des méthodes directes pour trouver une telle représentation par une union d'ordre de contiguïté (non nécessairement en nombre minimal) sans passer par un réalisateur (ce qui est difficile à calculer dans le cas général) ?

1.3 Reconnaissance des ordres N-free

La théorie des ordres N-free a été étudiée en profondeur pour leurs nombreuses propriétés structurelles [40, 39, 53, 45, 44]. Une de leurs plus anciennes et principales applications est leur utilisation dans l'analyse de projets, en particulier avec des techniques telles que CPM et PERT [25, 61]. Ces techniques représentent un projet par un graphe orienté dans lequel les arcs correspondent aux activités du projet et les sommets aux échéances (l'aboutissement de toutes les activités pointant sur le sommet). Dans le jargon de la théorie des ordres, cette représentation d'activités sous forme d'arcs, appelée réseau de PERT, est le diagramme d'arcs «*edge diagram* » d'un ordre N-free. Si l'ordre original décrivant les contraintes de précédences technologiques du projet n'est pas N-free, des activités factices sont ajoutées pour le rendre N-free. De nombreuses techniques ont été proposées pour cela [78, 79, 74].

L'autre application majeure des ordres N-free apparaît dans les recherches sur le nombre de sauts. Ce paramètre classique qui peut être calculé par un algorithme simple dans le cas des ordres N-free [70] intervient dans plusieurs propriétés structurelles des ordres N-free. Dans le cas général, le calcul de ce nombre est NP-difficile.

Les algorithmes séquentiels de reconnaissance des ordres N-free les plus rapides supposent que la réduction transitive de l'ordre est donnée et construisent un diagramme d'arcs si l'ordre est N-free. Ils s'exécutent en temps $O(n + m)$ où m est le nombre d'arcs de la réduction transitive de l'ordre. Le premier algorithme de ce type est implicitement contenu dans l'algorithme de reconnaissance des ordres série-parallèles de [83] (les ordres N-free peuvent être vus comme une généralisation des ordres série-parallèles). Le premier algorithme « explicite » linéaire de reconnaissance des ordres N-free est apparu dans [77]. Un autre résultat important a été apporté par MA et SPINRAD [55] qui ont donné un algorithme ne faisant aucune hypothèse sur la forme de l'ordre en entrée. Leur algorithme détermine si la fermeture transitive d'un graphe orienté sans circuit est un ordre N-free en temps $O(n + m_t)$ où m_t est le nombre d'arcs de la fermeture transitive de l'entrée.

Nous allons maintenant voir des algorithmes parallèles de reconnaissance des ordres N-free dans différents modèles PRAM, puis des algorithmes de construction d'un diagramme d'arcs en seront dérivés. Les algorithmes EREW s'exécutent en temps $O(\log n)$ avec $n + m$ processeurs et les algorithmes CRCW s'exécutent en temps constant avec n^2 processeurs.

Commençons tout d'abord par introduire les définitions et les propriétés structurelles qui nous seront nécessaires.

Définition et propriétés des ordres N-free

Un ordre est dit *N-free* si son diagramme de HASSE (c'est-à-dire sa réduction transitive) ne contient pas la structure interdite « N » de la figure 1.8 (a).

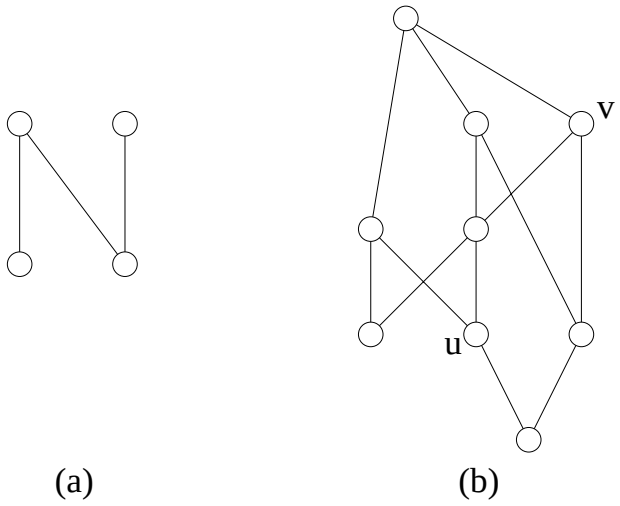


FIG. 1.8 – *(a) La sous-structure interdite pour les ordres N-free. (b) Un exemple d'ordre N-free représenté par son diagramme de Hasse.*

Les ordres N-free peuvent aussi être définis à partir d'une construction simple partant d'un graphe orienté sans circuit quelconque. Tout graphe orienté sans circuit $D = (V, \mathcal{A})$ induit un ordre $P = (\mathcal{A}, <)$ sur ses *arcs* de la manière suivante : $a < b$ si et seulement si il existe un chemin orienté de la destination de a à l'origine de b dans D. b couvre a si et seulement si la destination de a est l'origine de b. Le graphe orienté sans circuit d'ensemble de sommets $\mathcal{A}$ induit par la relation de couverture de P est souvent appelé en anglais *line-graph* de D. Un tel ordre P est dit *arc-induit* «*edge-induced* » et D est un *diagramme d'arcs* «*edge diagram* » de P.

Le théorème suivant [62] donne les propriétés structurelles fondamentales des ordres N-free nécessaires à la reconnaissance en séquentiel.

Théorème 9 ([62]) *Etant donné un ordre $P = (V, <)$, les propositions suivantes sont équivalentes :*

(1.1) *P est N-free.*

(1.2) *Pour tout $u, v \in V$, $ImSucc(u) = ImSucc(v)$ ou $ImSucc(u) \cap ImSucc(v) = \emptyset$.*

(1.3) *P est arc-induit.*

L'algorithme séquentiel de reconnaissance [77] vérifie la propriété (1.2) en traitant incrémentalement chaque sommet et l'ensemble de ses successeurs immédiats.

Une approche équivalente a été développée dans [83] et [27]. Nous verrons qu'elle est plus appropriée à l'algorithmique parallèle. Elle repose sur la définition des graphes orientés sans circuit *composés de bipartis complets* (graphes orientés sans circuit CBC en abrégé, voir la figure 1.9) qui sont les graphes orientés sans circuit $D = (V, \mathcal{A})$ admettant une partition $\mathcal{B}_1, \ldots, \mathcal{B}_k$ de leur ensemble d'arcs $\mathcal{A}$ telle que :

(2.1) Chaque $\mathcal{B}_i$ induit un sous-graphe de D biparti et complet (voir la figure 1.9), $\mathcal{B}_i$ étant appelée une *composante bipartie* de D.

(2.2) Pour tout sommet v autre qu'un puits, tous les arcs sortants de v appartiennent à la même composante bipartie.

(2.3) Pour tout sommet v autre qu'une source, tous les arcs entrants dans v appartiennent à la même composante bipartie.

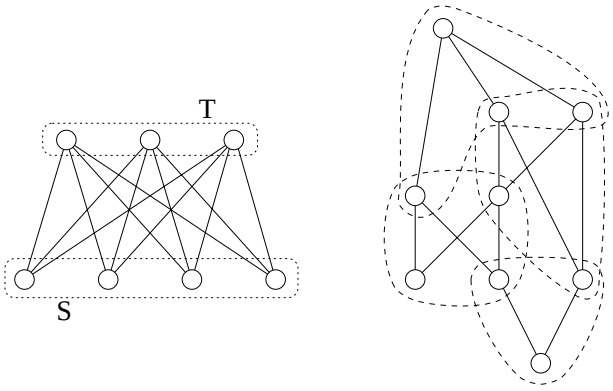


FIG. 1.9 – *(a) Un graphe biparti complet d'ensemble de sources S et d'ensemble de puits T. (b) Les composantes biparties de la réduction transitive de l'ordre N-free de la figure 1.8.*

Les assertions (2.2) et (2.3) sont simplement une condition de maximalité des composantes biparties. Grâce à cette définition, un autre théorème de caractérisation est donné :

Théorème 10 (Valdes, Tarjan, Lawler [83]) *Un graphe orienté sans circuit est* CBC *si et seulement s'il est la réduction transitive d'un ordre N-free.*

Notons respectivement S_i et T_i l'ensemble des sources et l'ensemble des puits de la composante bipartie $\mathcal{B}_i$. Alors on a pour tout $v \in T_i$, $ImPred(v) = S_i$ et

pour tout $u \in S_i$, $ImSucc(u) = T_i$, d'où la remarque suivante :

$$\{S_i,\ 1 \leq i \leq k\} = \{ImPred(v),\ v \in V\}$$
$$\text{et} \qquad \{T_i,\ 1 \leq i \leq k\} = \{ImSucc(u),\ u \in V\}.$$

Chaque S_i (respectivement T_i) sera appelé *ensemble source de composante* (respectivement *ensemble puits de composante*) dont la composante bipartie est $\mathcal{B}_i$.

Algorithmes parallèles de reconnaissance

Nous pouvons maintenant exhiber deux algorithmes parallèles de reconnaissance ; ils sont tous les deux basés sur le théorème 10 et ils calculent les composantes biparties sous l'hypothèse que l'entrée est donnée sous forme transitivement réduite. Ils sont deux réalisations différentes de l'algorithme général suivant 1.10 qui est indépendant de la structure de données.

Algorithme 1.10 Reconnaissance des ordres N-free

Données : Un graphe orienté sans circuit $D = (V, \mathcal{A})$ transitivement réduit.

Résultat : *Vrai* si la fermeture transitive de D est un ordre N-free, *Faux* sinon.

Etape 1 *Calculer les composantes biparties en supposant que D est la réduction transitive d'un ordre N-free comme suit.*

Sélectionner un sommet u_i dans chaque ensemble source de composante S_i en utilisant $\{S_i,\ 1 \leq i \leq k\} = \{ImPred(v),\ v \in V\}$.

Pour tout *sommet u_i sélectionné* **effectuer**

Identifier $T_i = ImSucc(u_i)$.

Identifier $\mathcal{B}_i$ comme l'ensemble des arcs dont la destination appartient à T_i.

Etape 2 *Vérifier que ce sont bien des composantes biparties.*

Vérifier si les trois conditions (2.2), (2.3) et (2.1) des graphes orientés sans circuit CBC sont valides. Si c'est le cas, D est la réduction transitive d'un ordre N-free ; sinon, retourner **Faux**.

Remarquons que les composantes calculées à l'étape 1 forment dans tous les cas une partition de l'ensemble des arcs. De plus ces composantes induisent des sous-graphes bipartis de D car D est transitivement réduit. Si ce n'était pas le cas, une composante $\mathcal{B}_i$ contiendrait deux arcs de la forme uv et vw. Par construction, elle contiendrait aussi les arcs u_iv et u_iw, ce dernier se trouvant être alors un arc de transitivité, ce qui est interdit. Pour la vérification de la condition (2.1) à l'étape 2, il suffit donc de vérifier que ces sous-graphes sont complets.

Remarquons aussi que tous les arcs entrant dans un sommet $v \in T_i$ seront dans la même composante par construction. La condition (2.3) est donc toujours valide et n'a pas besoin d'être testée. D'un autre côté, quand on vérifie avec succès

que tous les arcs sortant d'un sommet u appartiennent à la même composante bipartie (condition (2.2)), on sait alors que $u \in S_i$. Les ensembles sources de composantes sont donc calculés du même coup.

Précisons enfin comment seront maniés les sous-ensembles disjoints. Les composantes biparties sont des sous-ensembles disjoints de $\mathcal{A}$. Chaque $\mathcal{B}_i$ sera numéroté u_i. Les composantes biparties seront représentées par un tableau $\underline{\mathcal{B}}$ tel qu'un arc a est dans $\mathcal{B}_i$ si et seulement si $\underline{\mathcal{B}}[a] = u_i$. Les S_i (respectivement les T_i) sont des sous-ensembles disjoints de V. Chaque S_i (respectivement T_i) aura même numéro que $\mathcal{B}_i$. Les S_i (respectivement les T_i) seront représentés de manière similaire par un tableau $\underline{S}$ (respectivement $\underline{T}$).

Théorème 11 *L'algorithme 1.10 détermine si un graphe orienté sans circuit D réduit transitivement a pour fermeture transitive un ordre N-free.*

La preuve découle du théorème 10. Si l'entrée D est la réduction d'un ordre N-free (c'est-à-dire un graphe orienté sans circuit CBC), alors l'algorithme calcule effectivement ses composantes biparties à l'étape 1 et les trois tests seront réussis à l'étape 2.

Quelle que soit la partition calculée à l'étape 1, si les trois tests réussissent à l'étape 2, D est un graphe orienté sans circuit CBC. Si D n'est pas la réduction transitive d'un ordre N-free, alors l'algorithme n'a pas pu calculer des composantes biparties à l'étape 1 et l'un des tests de l'étape 2 échouera.

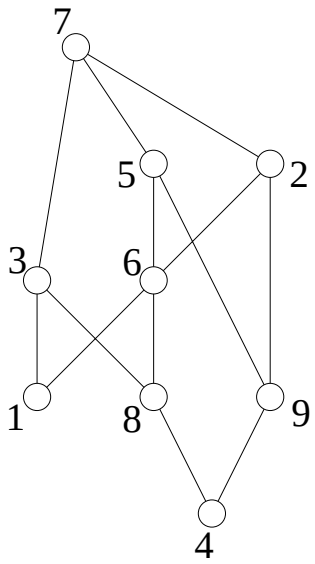
Nous pouvons maintenant proposer un algorithme EREW.

Un algorithme en lecture et écriture exclusives

Dans les algorithmes qui suivent, on identifie les sommets et leur numéro, plus formellement, on suppose que $V = \{1, \ldots, n\}$. Le premier algorithme de reconnaissance utilise un tableau d'arcs $\underline{\mathcal{A}}$ comme structure de données. Les arcs sont explicitement stockés sous forme de couples de sommets dans $\underline{\mathcal{A}}$. Cet algorithme est basé sur des tris du tableau des arcs dans l'ordre lexicographique ou dans l'ordre anti-lexicographique en utilisant l'ordre total sur les sommets donnés par leur numéro $1 < 2 < \cdots < n$ (voir la figure 1.10).

Dans cette réalisation de l'algorithme 1.10, le sommet choisi dans chaque ensemble source de composante sera celui de plus petit numéro. L'idée est de sélectionner ces éléments $u_1, \ldots, u_k$ sans avoir calculé les ensembles sources de composantes $S_1, \ldots, S_k$ qui les contiennent respectivement. Pour cela, il suffit de trier anti-lexicographiquement le tableau d'arcs. Considérons les arcs d'un bloc ayant même destination v. En ne gardant que leurs origines, on obtient la liste triée des éléments de $\mathit{ImPred}(v)$. Le premier élément sera évidemment le même pour tout w tel que $\mathit{ImPred}(w) = \mathit{ImPred}(v)$. Dans l'exemple de la figure 1.10, les sommets 6, 1, 2 et 4 seront sélectionnés. Grâce à ce tri, il est ensuite facile d'identifier le T_i et le $\mathcal{B}_i$ correspondants et de leur donner le numéro u_i.

(*a*)



(*b*) | (1,3) (1,6) | (2,7) | (3,7) | (4,8) (4,9) | (5,7) | (6,2) (6,5) | (8,3) (8,6) | (9,2) (9,5) |

(*c*) | (6,2) (9,2) | (1,3) (8,3) | (6,5) (9,5) | (1,6) (8,6) | (2,7) (3,7) (5,7) | (4,8) | (4,9) |

FIG. 1.10 – (*a*) *L'ordre N-free de la figure 1.8 avec les sommets numérotés (arbitrairement).* (*b*) *La représentation de sa réduction transitive sous forme de tableau d'arcs trié lexicographiquement. Les blocs de couples ayant même origine sont entourés.* (*c*) *Le tableau d'arcs trié anti-lexicographiquement. Les blocs de couples ayant même destination sont entourés.*

Nous vérifierons que chaque composante $\mathcal{B}_i$ calculée à l'étape 1 est complète en nous assurant que toutes ses sources ont même degré sortant. Ceci suffit car $\widehat{Degré}^+(u_i) = |T_i|$ par construction. Les degrés sortants peuvent être facilement obtenus en temps $O(\log n)$ avec $n + m$ processeurs en effectuant par exemple un tri lexicographique et un calcul préfixé.

Voir l'algorithme 1.11.

Théorème 12 *L'algorithme 1.11 détermine si un graphe orienté sans circuit transitivement réduit est N-free. Il fonctionne dans le modèle* EREW PRAM *en temps* $O(\log n)$ *avec* $n + m$ *processeurs.*

L'algorithme 1.11 est clairement équivalent à l'algorithme 1.10. La lecture concurrente à la ligne 1 peut être réalisée avec un calcul préfixé en temps $O(\log n)$ avec m processeurs. Les tris et les conjonctions prennent un temps $O(\log n)$ avec $n + m$ processeurs. Ceci prouve le résultat.

Algorithme 1.11 Reconnaissance des ordres N-free en EREW

Données : Un graphe orienté sans circuit $D = (V, \mathcal{A})$ transitivement réduit de tableau d'arcs $\underline{\mathcal{A}}$.

Résultat : *Vrai* si la fermeture transitive de D est un ordre N-free, *Faux* sinon.

Etape 1

Trier $\underline{\mathcal{A}}$ dans l'ordre anti-lexicographique.

Pour tout $1 \leq j \leq m$ **effectuer**

Soit uv l'arc en position $\underline{\mathcal{A}}[j]$.

Si uv est le premier arc de son bloc (c'est-à-dire si la destination de l'arc en position $\underline{\mathcal{A}}[j-1]$ est différente de v) alors poser $\underline{T}[v] \leftarrow u$.

1 Assigner à l'arc uv le numéro de composante bipartie $\underline{\mathcal{B}}[j] \leftarrow \underline{T}[v]$.

Etape 2

Vérification de la condition (2.2) : tous les arcs sortant d'un sommet sont-ils dans la même composante bipartie ?

Trier $\underline{\mathcal{A}}$ lexicographiquement.

Pour tout $2 \leq j \leq m$ **effectuer**

Soient uv et xy les arcs en position respective $\underline{\mathcal{A}}[j-1]$ et $\underline{\mathcal{A}}[j]$.

Si $u = x$ **Alors**

ValeurRetour$[j] \leftarrow (\underline{\mathcal{B}}[j-1] = \underline{\mathcal{B}}[j])$

Sinon

ValeurRetour$[j] \leftarrow$ Vrai

$\underline{S}[y] \leftarrow \underline{\mathcal{B}}[j]$

Soit uv l'arc en position $\underline{\mathcal{A}}[1]$, poser $\underline{S}[u] \leftarrow \underline{\mathcal{B}}[1]$.

Si $\bigwedge_{j=2}^{m}$ ValeurRetour$[j]$ = Faux, retourner le résultat Faux.

Vérification de la condition (2.1) : les composantes biparties sont-elles complètes ?

Ranger les couples $\underline{S}[u], \mathit{Degré}^+(u)$ $(1 \leq u \leq n)$ dans l'ordre lexicographique.

Vérifier de la même façon que précédemment qu'à l'intérieur de chaque bloc, tous les couples ont même deuxième composante.

Si les deux tests ont réussi, retourner le résultat Vrai.

Un algorithme en temps constant

Nous allons maintenant voir un algorithme en temps constant. Cela est possible dans le modèle CRCW PRAM en utilisant la puissance des écritures concurrentes arbitraires à la place des tris pour calculer des partitions. Nous vérifierons que les composantes biparties sont complètes grâce au graphe complémentaire en testant si aucun arc ne manque à la composante. L'utilisation du complémentaire impose un travail en $O(n^2)$. L'algorithme 1.12 est basé sur une représentation de

l'entrée sous la forme d'une matrice d'adjacence $\underline{M}$: $\underline{M}[u, v]$ = Vrai si et seulement si l'arc uv est dans $\mathcal{A}$. Il est donc facile de calculer le complémentaire en temps constant avec n^2 processeurs.

Pour réaliser l'étape 1, les composantes biparties seront calculées en deux phases. Considérons un ensemble source de composante S_i. Pour tout $v \in V$ tel que $\mathit{ImPred}(v) = S_i$, il faut isoler le même sommet u_i. On commence par sélectionner arbitrairement un $\underline{U}[v] \in \mathit{ImPred}(v)$ pour chacun des ces v, ce qui donne plusieurs sommets marqués dans S_i ; on ne garde que u_i, celui de numéro minimal. T_i est alors donné par $\mathit{ImSucc}(u_i)$ et $\mathcal{B}_i$ est l'ensemble des arcs entrant dans T_i. La figure 1.11 illustre cette procédure sur un exemple. Voir aussi l'algorithme 1.12.

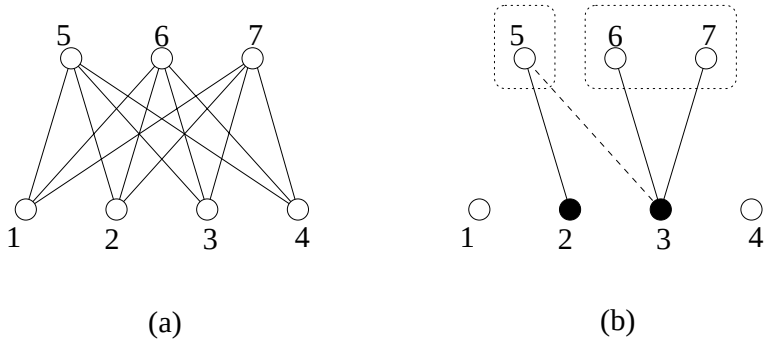


FIG. 1.11 – *Sélection d'une source dans chaque composante bipartie. (a) Une composante bipartie dont les sommets sont numérotés. (b) Une écriture concurrente (arbitraire) a donné* $\underline{U}[5] = 2$ *et* $\underline{U}[6] = \underline{U}[7] = 3$. 2 *et* 3 *sont donc marqués. L'existence de l'arc* $3, 5$ *prouve que* 3 *et* $2 = \underline{U}[5]$ *sont dans le même ensemble source de composante* $\mathrm{ImPred}(5) = \mathrm{ImPred}(6) = \mathrm{ImPred}(7)$. *Comme* $2 < 3$, *la marque de* 3 *est effacée et* 2 *reste la seule source marquée de la composante.*

Théorème 13 *L'algorithme 1.12 détermine si un graphe orienté sans circuit transitivement réduit est N-free. Il fonctionne dans le modèle* CRCW PRAM *en temps constant avec* n^2 *processeurs.*

Preuve. Les bornes en temps et en nombre de processeurs sont claires. La preuve de l'exactitude de l'algorithme est moins évidente.

La vérification de la condition (2.2) se fait en vérifiant que l'écriture concurrente arbitraire $\underline{S}[u] \leftarrow \underline{\mathcal{B}}[u, v]$ était en fait une écriture concurrente de la même valeur. Remarquons qu'une fois ce test vérifié, deux sommets u et v sont dans le même ensemble source de composante si et seulement si $\underline{S}[u] = \underline{S}[v]$. On vérifie ensuite que chaque composante bipartie $\mathcal{B}_i$ est bien constituée de tous les arcs de S_i vers T_i (c'est-à-dire la condition (2.1)) en s'assurant qu'aucun arc uv avec $u \in S_i$ et $v \in T_i$ ne manque dans le graphe orienté sans circuit.

Algorithme 1.12 Reconnaissance des ordres N-free en CRCW

Données : Un graphe orienté sans circuit $D = (V, \mathcal{A})$ transitivement réduit de matrice d'adjacence $\underline{M}$.

Résultat : *Vrai* si la fermeture transitive de D est un ordre N-free, *Faux* sinon.

Etape 1

Pour tout $1 \leq u, v \leq n$ *tels que* $\underline{M}[u,v] = Vrai$ **effectuer**
  $\underline{U}[v] \leftarrow u$ {*écriture concurrente arbitraire*}

Pour tout $1 \leq u \leq n$ **effectuer** Marqué$[u] \leftarrow$ Faux
Pour tout $1 \leq v \leq n$ **effectuer** Marqué$[\underline{U}[v]] \leftarrow$ Vrai
Pour tout $1 \leq u, v \leq n$ *tels que* $\underline{M}[u,v] = Vrai$ *et* $u > \underline{U}[v]$ **effectuer**
  Marqué$[u] \leftarrow$ Faux {*écriture concurrente de la même valeur*}

Pour tout $1 \leq u, v \leq n$ *tels que* $\underline{M}[u,v] = Vrai$ **effectuer**
  Si *Marqué*$[u] = Vrai$ **Alors** $\underline{T}[v] \leftarrow u$
  $\underline{\mathcal{B}}[u,v] \leftarrow \underline{T}[v]$

Etape 2

Résultat $\leftarrow$ Vrai
Vérification de la condition (2.2) : tous les arcs sortant d'un sommet sont-ils dans la même composante bipartie ?
  Pour tout $1 \leq u, v \leq n$ *tels que* $\underline{M}[u,v] = Vrai$ **effectuer**
    $\underline{S}[u] \leftarrow \underline{\mathcal{B}}[u,v]$ {*écriture concurrente arbitraire*}
    Si $\underline{S}[u] \neq \underline{\mathcal{B}}[u,v]$ **Alors**
      Résultat $\leftarrow$ Faux {*vérifier que c'est en fait la même valeur qui vient d'être écrite concurrenciellement*}

Vérification de la condition (2.1) : les composantes biparties sont-elles complètes ?
  Pour tout $1 \leq u, v \leq n$ *tels que* $\underline{M}[u,v] = Faux$ **effectuer**
    Si $\underline{S}[u]$ *et* $\underline{T}[v]$ *sont définis et* $\underline{S}[u] = \underline{T}[v]$ **Alors** Résultat $\leftarrow$ Faux

Retourner Résultat.

Il reste à prouver que la réalisation de l'étape 1 est fidèle à l'algorithme 1.10. Nous supposons pour cela que l'entrée est CBC, si ce n'est pas le cas, ce qui est calculé n'a pas d'importance.

Considérons une composante bipartie $\mathcal{B}_i$ constituée de tous les arcs de S_i vers T_i. Pour tout $v \in T_i$, $U[v]$ est un sommet arbitraire dans $S_i = ImPred(v)$. Les sommets de la forme $U[v]$ sont marqués, soit u_i le sommet marqué de S_i de numéro minimal et v_i un sommet tel que $U[v_i] = u_i$. Si u est un autre sommet marqué dans S_i, l'arc uv_i effacera sa marque. Les arcs sortant de u_i étant internes à la composante bipartie, sa marque ne sera pas effacée. Le reste de l'étape 1 est clairement fidèle à l'algorithme 1.10. □

Construction d'un diagramme d'arcs

En étudiant les propriétés des composantes biparties, nous allons voir comment on peut facilement modifier les algorithmes de reconnaissance pour qu'ils construisent un diagramme d'arcs induisant un ordre N-free donné.

Considérons la réduction transitive $D = (V, \mathcal{A})$ d'un ordre N-free P. D est un graphe orienté sans circuit CBC et notons encore $\mathcal{B}_1, \ldots, \mathcal{B}_k$ ses composantes biparties, S_i l'ensemble des sources de $\mathcal{B}_i$ et T_i l'ensemble des puits de $\mathcal{B}_i$.

Notons T_0 l'ensemble des sources de D et S_∞ l'ensemble de ses puits. Aussi, $T_0, T_1, \ldots, T_k$ et $S_1, \ldots, S_k, S_\infty$ sont deux partitions de V. Nous introduisons de plus deux composantes biparties « virtuelles » $\mathcal{B}_0$ et $\mathcal{B}_\infty$ en posant formellement que l'ensemble puits de $\mathcal{B}_0$ est T_O et que l'ensemble source de $\mathcal{B}_\infty$ est S_∞.

Considérons maintenant le graphe orienté sans circuit $A = (\{\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_k, \mathcal{B}_\infty\}, \widetilde{V})$ défini comme suit (voir aussi la figure 1.12). Chaque sommet $v \in V$ est associé à un arc $\widetilde{v} \in \widetilde{V}$ de A :

si $v \in T_i$ et $v \in S_j$ alors $\widetilde{v} = \mathcal{B}_i\mathcal{B}_j$ est un arc de $\mathcal{B}_i$ vers $\mathcal{B}_j$.

A peut avoir des arcs multiples lorsqu'il y a plusieurs sommets dans $T_i \cap S_j$ ($\widetilde{V}$ est un multi-ensemble).

Théorème 14 ([83]) *Le graphe orienté sans circuit A est un diagramme d'arcs de P.*

Ceci vient du fait que D est le line-graph de A puisque ses arcs uv sont ceux vérifiant $u \in S_i$ et $v \in T_i$ pour un certain $i \in \{1, \ldots, k\}$ et on peut écrire $u = \mathcal{B}_h\mathcal{B}_i$ et $v = \mathcal{B}_i\mathcal{B}_j$ où $h, j \in \{0, 1, \ldots, k, \infty\}$.

Remarquons que A est le seul diagramme d'arcs de P qui n'a qu'une seule source et un seul puits.

Comme les algorithmes proposés pour la reconnaissance des ordres N-free calculent les S_i et les T_i, il ne reste pas beaucoup de travail à effectuer pour obtenir un diagramme d'arcs de l'ordre. Supposons que l'on initialise au début de l'algorithme le tableau $\underline{S}$ à 0 et le tableau $\underline{T}$ à ∞. Les sources (respectivement les puits) sont les seuls sommets pour lesquels $\underline{T}$ (respectivement $\underline{S}$) n'est pas assigné. Par conséquent, à la fin de l'algorithme, le tableau $\underline{T}$ (respectivement $\underline{S}$) représentera la partition $\{T_0, T_1, \ldots, T_k\}$ (respectivement $\{S_1, \ldots, S_k, S_\infty\}$) où T_0 est numéroté 0 (et S_∞ est numéroté ∞).

En ajoutant la ligne suivante à la fin des algorithmes précédents, on obtient le diagramme d'arcs A de l'ordre N-free sans en changer les complexités :

Pour tout $1 \leq u \leq n$ effectuer $\underline{\widetilde{V}}[u] \leftarrow \underline{T}[v], \underline{S}[v]$.

Théorème 15 *Les algorithmes 1.11 et 1.11 augmentés de la ligne ci-dessus déterminent si un ordre est N-free étant donnée sa réduction transitive et construisent si c'est le cas un diagramme d'arcs associé. Ils fonctionnent respectivement en temps $O(\log n)$ avec $n + m$ processeurs d'une* EREW PRAM *et en temps constant avec n^2 processeurs d'une* CRCW PRAM.

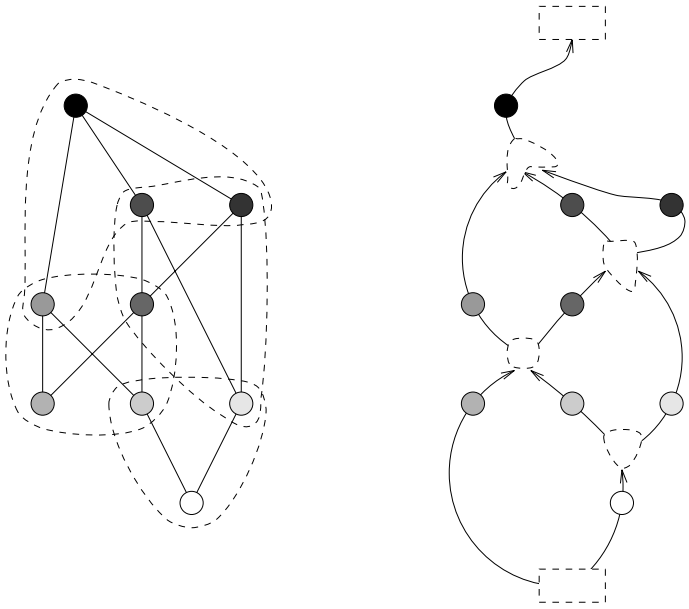


FIG. 1.12 – *(a) L'ordre N-free de la figure 1.8 représenté par son diagramme de* HASSE *où les composantes biparties sont entourées. (b) Son unique diagramme d'arcs ne possédant qu'une seule source et un seul puits. Ses sommets sont les composantes biparties de l'ordre N-free (plus une source et un puits). Chaque arc est associé à un sommet de l'ordre N-free.*

La preuve découle du théorème 14.

Remarque. Un diagramme d'arcs A est une représentation sublinéaire de la représentation de l'ordre N-free P puisque sa taille est $O(n)$. Cette représentation permet de répondre en tant constant à une requête du type « u est-il un prédécesseur immédiat de v? » en testant si la destination $\widetilde{u}$ est l'origine de $\widetilde{v}$.

La fermeture transitive de P peut-être déduite de la fermeture transitive de A puisque l'on peut répondre à la requête « u est-il un prédécesseur de v? » en testant s'il existe dans A un chemin orienté de la destination de $\widetilde{u}$ à l'origine de $\widetilde{v}$. Ceci est intéressant puisqu'en général A est beaucoup plus petit que la réduction transitive de P.

En ce qui concerne l'algorithme CRCW, l'utilisation du graphe complémentaire ne sert qu'à vérifier si les composantes biparties sont bien complètes. Si l'on se passe du test, l'algorithme peut calculer un diagramme d'arcs en temps constant avec $n + m$ processeurs si l'entrée est supposée être la réduction transitive d'un ordre N-free. Si l'on fournissait une entrée non N-free à cet algorithme, il le détecterait qu'elle n'est pas N-free (en vérifiant la condition (2.2)) ou fournirait une extension N-free minimale (en complétant les composantes biparties non

complètes).

Algorithmes pour des machines distribuées

L'algorithme 1.11 est composé de tris et de calculs préfixés. Ces procédures ayant été intensivement étudiés pour les différentes architectures de machines distribuées, on peut aisément en déduire des algorithmes distribués pour reconnaître les ordres N-free et construire leurs diagrammes d'arcs. Voici quelques exemples.

Considérons une architecture d'hypercube. Le tri de [16] tourne en temps $O(n \log n (\log \log n)^2)/p$ avec p processeurs et le calcul de sommes préfixées de [64] tourne optimalement en temps $O(\log n)$. L'algorithme 1.11 peut être implanté sur un hypercube à p processeurs et tourner alors en temps $O((n + m) \log n (\log \log n)^2/p)$.

Sur une grille $\sqrt{p} \times \sqrt{p}$, les sommes préfixées sont implantables optimalement (pour la grille, cela veut dire en temps $O(\sqrt{p})$), ainsi que le tri circulaire de [56] ou le tri bitonic de [4]. L'algorithme 1.11 peut donc être implanté sur une grille $\sqrt{p} \times \sqrt{p}$ pour s'exécuter optimalement en temps $O(\sqrt{p})$.

Nous avons vu comment effectuer la reconnaissance des ordres N-free en utilisant essentiellement des routines parallèles élémentaires, ce qui rend ces algorithmes portables sur différentes architectures. Nous allons maintenant aborder un problème de reconnaissance plus complexe, celui des graphes de comparabilité.

1.4 Reconnaissance des graphes de comparabilité et décomposition modulaire

Dans [37], Golumbic développe une théorie algorithmique des graphes de comparabilité. Son algorithme d'orientation transitive a été amélioré séparément par Cournier et Habib [15] et par Mcconnel et Spinrad [57]. Les relations entre les graphes de comparabilité et la décomposition modulaire ont été découvertes par Gallai [34].

Nous allons reprendre ici la démarche de Golumbic dans le cadre de l'algorithmique parallèle. Si l'algorithme de reconnaissance se parallélise assez naturellement, ce n'est pas le cas pour celui d'orientation transitive. Il faut pour cela considérer d'autres propriétés structurelles des graphes de comparabilité. Il se trouve que ces structures sont les multiplexes maximaux introduits par Golumbic dans un tout autre but : compter le nombre d'orientations transitives. En approchant cette structure sous l'angle du parallélisme nous découvrirons les relations intéressantes qu'elle entretient avec la décomposition modulaire.

Avant de proposer une parallélisation de l'algorithme de reconnaissance de Golumbic, il nous faudra rappeler les rudiments de la théorie de Gallai. Nous étendrons ensuite les résultats de Golumbic sur les multiplexes maximaux pour

en déduire un algorithme d'orientation transitive. Nous ferons enfin le lien avec la décomposition modulaire, en déduisant au passage un algorithme parallèle de décomposition modulaire. Ces trois algorithmes sont limités par le calcul des classes de forçage qui demande un temps $O(\log n)$ avec δm processeurs d'une CRCW PRAM où δ est le degré maximal du graphe d'entrée.

La théorie de Gallai

Un graphe $G = (V, \mathcal{E})$ est un *graphe de comparabilité* s'il existe un ordre $P = (V, \mathcal{F})$ dont il est le graphe de comparabilité. $\mathcal{F}$ est alors appelé une orientation transitive des arêtes du graphe. Remarquons que $\mathcal{F}^{-1}$ est aussi une orientation transitive. C'est une orientation des arêtes dans le sens où $\mathcal{F} \cap \mathcal{F}^{-1} = \emptyset$ et $\mathcal{E} = \mathcal{F} \cup \mathcal{F}^{-1}$. Une orientation $\mathcal{F}$ des arêtes est transitive si elle vérifie de plus : $xy, yz \in \mathcal{F} \implies xz \in \mathcal{F}$.

On dira qu'une arête *touche* un sommet s'il en est l'une des extrémités. Si $\mathcal{A} \subseteq \mathcal{E}$ est un ensemble d'arcs, $V_{\mathcal{A}}$ désignera l'ensemble des sommets touchés par un arc de $\mathcal{A}$.

La théorie des graphes de comparabilité tourne autour de ce que Golumbic appelle un triangle, c'est-à-dire trois sommets a, b, c tels que les arêtes $\widehat{ab}, \widehat{bc}, \widehat{ca}$ sont présentes dans le graphe. C'est la présence de certains triangles spéciaux qui rendent l'orientation transitive difficile. En revanche, si l'une des arêtes manque, les deux autres ne peuvent pas être orientées n'importe comment, cela impose des contraintes qui se calculent facilement. Supposons $\widehat{ab}, \widehat{bc} \in \mathcal{E}$ et $\widehat{ac} \notin \mathcal{E}$. Il n'est pas possible d'orienter $\widehat{ab}$ par ab et $\widehat{bc}$ par bc car il manquerait alors l'arc de transitivité ac. On dit alors que *ab force directement cb* (de manière symétrique ba force directement bc) et on note $ab \sim cb$. Voir la figure 1.13 pour un exemple.

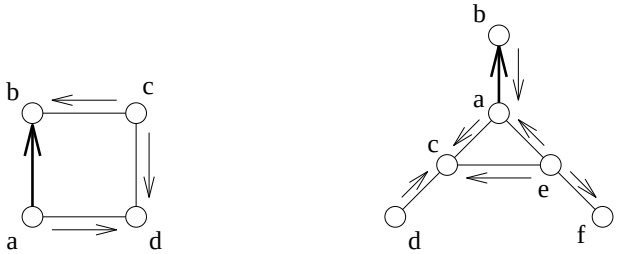


Fig. 1.13 – *Exemples de forçages. Le choix arbitraire de ab comme orientation de $\widehat{ab}$ force les autres orientations indiquées. Dans (ii), on a $ab \sim ac \sim cd \sim ce \sim fe \sim ea \sim ba$. Ceci mène à une contradiction puisqu'on ne peut pas orienter $\widehat{ab}$ avec ab et ba à la fois. Ce graphe n'est donc pas un graphe de comparabilité.*

Cette relation est symétrique et réflexive. Les classes d'équivalence de sa fer-

meture transitive $\overset{*}{\sim}$ sont appelées *classes d'implication* et forment une partition de l'ensemble des arcs. On dit que *ab force cd* quand $ab \overset{*}{\sim} cd$. De manière symétrique, on a alors $ba \overset{*}{\sim} dc$, de sorte que si $\mathcal{I}$ est une classe d'implication, $\mathcal{I}^{-1}$ en est une aussi. $\widehat{\mathcal{I}}$ est appelé *classe de couleur*. Les classes de couleur forment une partition de l'ensemble des arêtes (voir la figure 1.14).

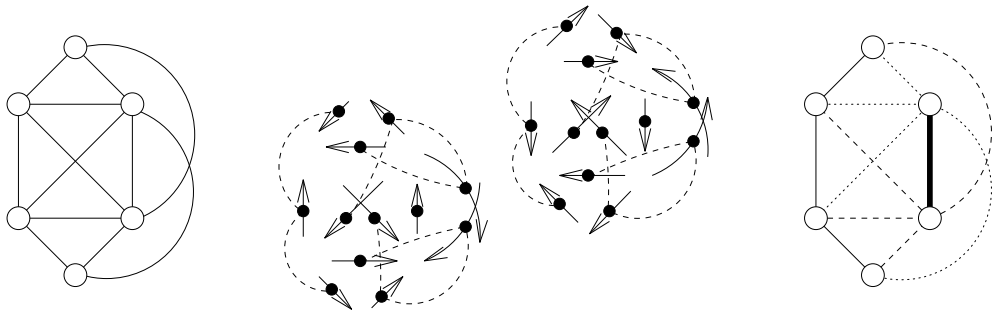


FIG. 1.14 – *(i) Un graphe non orienté $G = (V, \mathcal{E})$. (ii) Le graphe $(\mathcal{E}, \sim)$. Les lignes en pointillés représentent les relations de forçage direct (une arête correspond à deux arcs). Les classes d'implication de G sont les composantes connexes de ce graphe. (iii) Les classes de couleur de G.*

Remarquons que dans un graphe de comparabilité, un arc ab et son inverse ba ne sont jamais dans la même classe d'implication (voir la figure 1.13). De manière équivalente, chaque classe d'implication $\mathcal{I}$ est disjointe de son inverse $\mathcal{I}^{-1}$. L'inverse est vrai aussi comme le spécifie le théorème suivant.

Théorème 16 (Gallai [34]) *1. Si $\mathcal{I}$ est une classe d'implication, $\mathcal{I} = \mathcal{I}^{-1}$ ou $\mathcal{I} \cap \mathcal{I}^{-1} = \emptyset$.*

2. Si $\mathcal{I} \cap \mathcal{I}^{-1} = \emptyset$, alors $(V_{\widehat{\mathcal{I}}}, \widehat{\mathcal{I}})$ possède exactement deux orientations transitives : $\mathcal{I}$ et $\mathcal{I}^{-1}$.

3. Un graphe est de comparabilité si chacune de ses classes d'implication vérifie $\mathcal{I} \cap \mathcal{I}^{-1} = \emptyset$.

Le graphe de la figure 1.14 par exemple est donc un graphe de comparabilité.

Ce théorème conduit à un algorithme simple pour tester si un graphe est de comparabilité : calculer les classes d'implication et vérifier qu'aucun arc n'est dans la même classe que son inverse. Mais l'algorithme de GOLUMBIC n'est pas une simple formulation algorithmique du théorème de GALLAI. La partie difficile de l'algorithme réside dans le calcul d'une orientation transitive.

Nous pouvons d'ores et déjà donner la parallélisation de cette partie facile de l'algorithme de GOLUMBIC et nous aborderons ensuite les problèmes qui surgissent lors de l'orientation transitive. Cet algorithme parallèle utilise une

CRCW PRAM car il utilise l'algorithme des composantes connexes. Voir l'algorithme 1.13.

Algorithme 1.13 Reconnaissance des graphes de comparabilité

Données : Les arcs d'un graphe symétrique $G = (V, \mathcal{E})$ donnés sous forme de tableau des arcs et sous forme de listes d'adjacence triées[5].

Résultat : *Vrai* si G est un graphe de comparabilité, *Faux* sinon.

Etape 1 *Calcul de la relation de forçage direct* $\sim$.

- **Pour tout** *arc* ab *et* c *voisin de* a **effectuer**
  - Tester (en temps $O(\log \delta)$) si c fait partie de la liste triée des voisins de b.
  - Si $\widehat{bc} \notin \mathcal{E}$, stocker en mémoire $ab \sim ac$ et $ba \sim ca$.

Etape 2 *Calcul des classes d'implication.*

- Ce sont les composantes connexes du graphe $(\mathcal{E}, \sim)$.

Etape 3 *Vérifier si* G *est un graphe de comparabilité.*

- **Pour tout** *arc* $e = ab$ **effectuer**
  - Lire (en temps $O(\log \delta)$) le numéro de la classe d'implication de l'arc inverse ba.
  - Donner à ab le numéro de classe d'implication de ab et de ba le plus grand comme numéro de classe de couleur.
  - **Si** $e = ab$ *et* $e^{-1} = ba$ *ont même numéro de classe d'implication* **Alors**
    - $C_e \leftarrow$ Faux
  - **Sinon**
    - $C_e \leftarrow$ Vrai
- Retourner $\bigwedge_{1 \leq e \leq m} C_e$.

Théorème 17 *L'algorithme 1.13 détermine si un graphe est de comparabilité. Il calcule dans tous les cas les classes d'implication et de couleur de* G. *Il s'exécute dans le modèle* CRCW PRAM *en temps* $O(\log n)$ *avec* δm *processeurs.*

La complexité de l'algorithme découle de celles des procédures de base qu'il utilise et son exactitude vient du théorème 16.

La théorie de Golumbic

L'orientation transitive est plus difficile que la reconnaissance car les classes de couleur ne peuvent pas être orientées indépendamment les unes des autres. Dans un graphe complet par exemple, chaque classe de couleur est réduite à une seule arête. En orientant les arêtes arbitrairement, on risque de créer un circuit.

(Remarquons cependant qu'un graphe complet est le graphe de comparabilité de n'importe quel ordre total sur ses sommets.)

Les classes de couleur interagissent dans ce qui s'appelle les *multiplexes maximaux*. GOLUMBIC les a introduites pour compter le nombre d'orientations transitives.

Définition 18 (Golumbic [37]) *Soit G un graphe non orienté. Un sous-graphe complet $(V_{\mathcal{S}}, \mathcal{S})$ sur $r+1$ sommets est appelé* simplexe *de* rang r *si les arêtes de $\mathcal{S}$ apparaissent dans des classes de couleur distinctes de G. Le* multiplexe *généré par un simplexe de rang r est le sous graphe $(V_{\mathcal{M}}, \mathcal{M})$ où $\mathcal{M} = \cup\mathcal{C}$ est l'union des classes de couleur $\mathcal{C}$ telles que $\mathcal{C} \cap \mathcal{S} \neq \emptyset$.*

Vis à vis de l'orientation transitive, un multiplexe se comporte comme un simplexe le générant qui s'oriente en choisissant un ordre total sur ses sommets. Un simplexe de rang 2 est appelé triangle *tricolore*.

GOLUMBIC a prouvé les propriétés suivantes :

1. Les simplexes générant le même multiplexe sont isomorphes et ont donc même rang k. Le multiplexe qu'ils génèrent est dit avoir rang k aussi.
2. Un multiplexe est *maximal* (pour l'inclusion) si et seulement s'il est généré par un simplexe maximal.
3. Deux multiplexes maximaux sont, soit disjoints, soit égaux.
4. Si $\mathcal{I}$ est une classe d'implication telle que $\mathcal{I} = \mathcal{I}^{-1}$, alors $\mathcal{I}$ est elle même un multiplexe maximal de rang 1.

Théorème 19 (Golumbic [37]) *Soit $G = (V, \mathcal{E})$ un graphe non orienté avec $\mathcal{E} = \mathcal{M}_1 + \cdots + \mathcal{M}_k$ où chaque $\mathcal{M}_i$ est un multiplexe maximal.*

1. *Si $\mathcal{F}$ est une orientation transitive de G, alors $\mathcal{F} \cap \mathcal{M}_i$ est une orientation transitive de $\mathcal{M}_i$.*
2. *Si $\mathcal{F}_1, \ldots, \mathcal{F}_k$ sont des orientations transitives de $\mathcal{M}_1, \ldots, \mathcal{M}_k$ respectivement, alors $\mathcal{F}_1 + \cdots + \mathcal{F}_k$ est une orientation transitive de G.*
3. *Si G est un graphe de comparabilité et r_i désigne le rang de $\mathcal{M}_i$, alors le nombre d'orientations transitives de G est $\prod_{1 \leq i \leq k} (r_i + 1)!$.*

GOLUMBIC conclut finalement que les multiplexes maximaux forment une partition de l'ensemble des arêtes et sont indépendants vis à vis de l'orientation transitive. Un graphe de comparabilité se comporte donc comme une collection de graphes complets disjoints.

Le lemme suivant joue un rôle central dans le théorie de GOLUMBIC. Comme nous l'utiliserons dans un contexte non orienté, en voici une version sur les arêtes. La figure 1.15 en illustre la preuve.

Lemme 20 (Lemme du triangle [37]) *Soient $\mathcal{A}$ et $\mathcal{B}$ deux classes de couleur distinctes d'un graphe non orienté $G = (V, \mathcal{E})$ et trois sommets a, b, c tels que $\widehat{ac} \in \mathcal{B}$ et $\widehat{bc} \in \mathcal{A}$. Alors les propriétés suivantes sont vérifiées.*

1. *L'arête $\widehat{ab}$ est présente dans le graphe, soit $\mathcal{C}$ sa classe de couleur.*
2. *Si $\mathcal{C} \neq \mathcal{A}$ et $\widehat{b'c'} \in \mathcal{A}$ alors $\widehat{ac'} \in \mathcal{B}$ ou $\widehat{ab'} \in \mathcal{B}$.*
3. *Si $\mathcal{C} \neq \mathcal{A}$ alors aucune arête de $\mathcal{A}$ ne touche a.*
4. *Si $\widehat{b'c'} \in \mathcal{A}$ et $a'c' \in \mathcal{B}$ alors $a'b' \in \mathcal{C}$.*

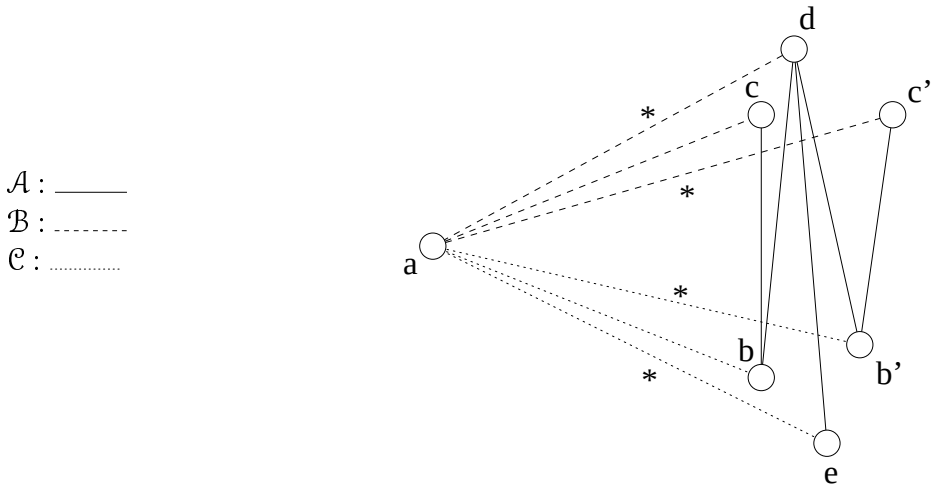


FIG. 1.15 – *Le lemme du triangle. L'existence et la couleur des arêtes marquées avec une $*$ vient des hypothèses suivantes : $\mathcal{B} \neq \mathcal{A}$ et $\widehat{bc}, \widehat{b'c'} \in \mathcal{A}$, et plus précisément $\widehat{bc} \sim \widehat{bd} \sim \widehat{ed} \sim \widehat{b'd} \sim \widehat{b'c'}$. De $\mathcal{C} \neq \mathcal{A}$, on déduit l'existence de l'arête $\widehat{ad}$. De $\widehat{cd} \notin \mathcal{E}$, on déduit $\widehat{ad} \in \mathcal{B}$. Le raisonnement est analogue pour les autres arêtes marquées.*

Extension de la théorie de GOLUMBIC

En séquentiel, une modification très simple de l'algorithme de reconnaissance permet de calculer une orientation transitive avec la même complexité. Les classes d'implication sont calculées les unes après les autres. En retirant les arêtes de la dernière classe calculée et en reprenant l'algorithme sur le graphe restant, GOLUMBIC obtient une « G-décomposition » où les classes sont des unions de classes de couleur du graphe originel et peuvent être orientées indépendamment les unes des autres pour obtenir une orientation transitive du graphe entier.

Malheureusement, cette approche gloutonne est intrinsèquement séquentielle. Pour résoudre ce problème en parallèle, il faut trouver une nouvelle approche algorithmique. Nous allons utiliser pour le faire une nouvelle vision des multiplexes

maximaux, et nous verrons même plus loin comment ils sont intimement liés à la décomposition modulaire.

La question cruciale est « Comment les classes de couleur interagissent-elles ? ». Soient $\mathcal{A}$ et $\mathcal{B}$ deux classes de couleur distinctes du graphe non orienté $G = (V, \mathcal{E})$. Nous dirons que $\mathcal{A}$ *touche* $\mathcal{B}$ quand $\mathcal{A} \cap \mathcal{B} \neq \emptyset$. En appliquant le lemme du triangle 20.4 deux fois dans chacun des trois cas $\mathcal{C} = \mathcal{B}$, $\mathcal{C} = \mathcal{A}$ et $\mathcal{A}, \mathcal{B}, \mathcal{C}$ distincts, on montre qu'il existe une unique classe de couleur $\mathcal{C}$ telle que pour tous $c \in V_{\mathcal{A}} \cap V_{\mathcal{B}}$, $\widehat{bc} \in \mathcal{A}$ et $\widehat{ac} \in \mathcal{B}$, l'arête $\widehat{ab}$ (qui doit alors exister dans G) est dans $\mathcal{C}$. On dira donc que $\mathcal{A}$ touche $\mathcal{B}$ *par* $\mathcal{C}$.

Si $\mathcal{A}$ touche $\mathcal{B}$ par $\mathcal{C}$, alors $\mathcal{C}$ touche $\mathcal{A}$ par $\mathcal{B}$ et $\mathcal{B}$ par $\mathcal{A}$. Si $\mathcal{C} = \mathcal{A}$, alors on est dans le cas où $V_{\mathcal{B}} \subset V_{\mathcal{A}}$ et on dira que $\mathcal{C}$ *couvre* $\mathcal{A}$. Si $\mathcal{A}, \mathcal{B}$ et $\mathcal{C}$ sont tous trois distincts, alors on est dans le cas où $V_{\mathcal{A}}$ et $V_{\mathcal{B}}$ sont incomparables par inclusion et on dira que $\mathcal{A}$ *croise* $\mathcal{B}$ (ou que $\mathcal{A}, \mathcal{B}, \mathcal{C}$ se croisent les uns les autres). Voir la figure 1.16.

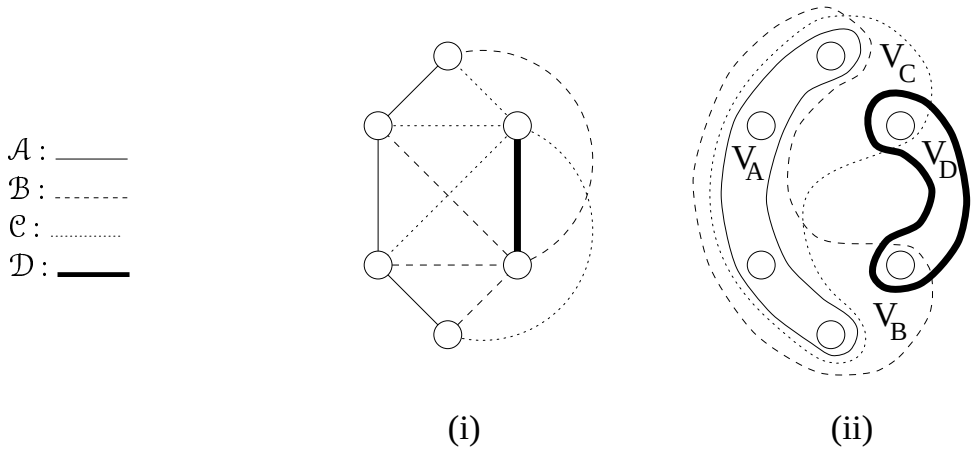


FIG. 1.16 – *(i) Le graphe de la figure 1.14. Ses classes de couleur sont $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$. (ii) Les classes de couleur $\mathcal{B}$ et $\mathcal{C}$ couvrent toutes les deux $\mathcal{A}$. Les classes de couleur $\mathcal{B}, \mathcal{C}, \mathcal{D}$ se croisent les unes les autres.*

Théorème 21 *Les unions maximales de classes de couleur se croisant l'une l'autre sont les multiplexes maximaux.*

Preuve. Un multiplexe est clairement une union de classes de couleur se croisant l'une l'autre. D'un autre côté, les propriétés des classes de couleur mises en évidence par le lemme du triangle 20 impliquent qu'une union de classes se croisant est incluse dans un multiplexe maximal (il est facile de mettre en évidence un simplexe l'engendrant). Le théorème est une conséquence de ces deux remarques. □

Cette nouvelle définition des multiplexes maximaux fournit un moyen simple de les calculer et de déterminer le nombre d'orientations transitives.

Remarquons qu'un multiplexe qui contient plus d'une classe de couleur en contient au moins trois. Ce théorème précise encore la forme de tels multiplexes.

Théorème 22 *Soit $\mathcal{M}$ un multiplexe maximal contenant au moins trois classes de couleur. Les assertions suivantes sont alors vérifiées.*

1. *Chaque classe de couleur $\mathcal{A} \subseteq \mathcal{M}$ induit un sous-graphe biparti complet dans le sens suivant. Soit $\mathcal{B}$ une classe croisant $\mathcal{A}$ par une classe $\mathcal{C}$. Alors $\mathcal{A}$ est l'ensemble des arêtes joignant un sommet de $V_\mathcal{A} \cap V_\mathcal{B}$ et un sommet de $V_\mathcal{A} \cap V_\mathcal{C}$. $V_\mathcal{A} \cap V_\mathcal{B}$ et $V_\mathcal{A} \cap V_\mathcal{C}$ sont qualifiés de* supersommets *de $\mathcal{M}$* reliés *par la* superarête *$\mathcal{A}$. $\mathcal{A}$ possède donc deux orientations transitives : l'ensemble des arcs de $V_\mathcal{A} \cap V_\mathcal{B}$ vers $V_\mathcal{A} \cap V_\mathcal{C}$ et son inverse.*
2. *Le graphe dont les sommets sont les supersommets de $\mathcal{M}$ et dont les arêtes sont ses superarêtes est un graphe complet (voir la figure 1.17). Tous les simplexes générant $\mathcal{M}$ peuvent être obtenus en prenant un sommet dans chaque supersommet de $\mathcal{M}$.*

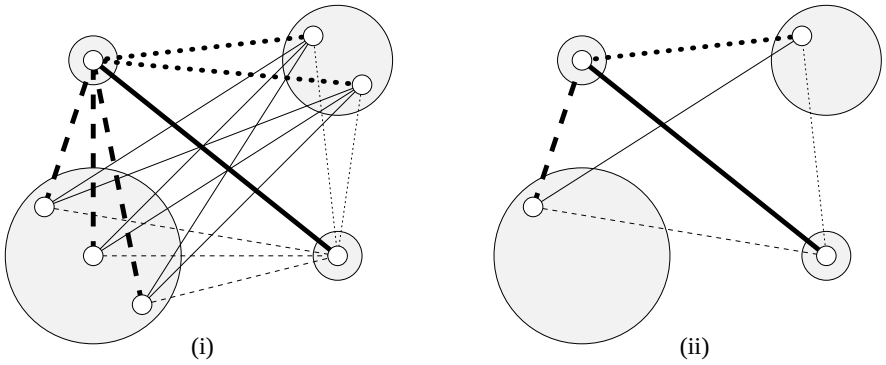


FIG. 1.17 – *(i) La structure de graphe complet d'un multiplexe maximal. Les disques gris représentent les supersommets. (ii) Un simplexe générant (i). Il est isomorphe au graphe complet sur les supersommets.*

Preuve. (1) Remarquons tout d'abord que le lemme du triangle 20 implique que $V_\mathcal{A} \cap V_\mathcal{B}$ et $V_\mathcal{A} \cap V_\mathcal{C}$ sont disjoints. Soit a, b, c un triangle tricolore avec $\widehat{bc} \in \mathcal{A}$ et $a \in V_\mathcal{B} \cap V_\mathcal{C}$, et soient $\mathcal{B}_a$ et $\mathcal{C}_a$ l'ensemble des arêtes de $\mathcal{B}$ et $\mathcal{C}$ respectivement touchant a. Le lemme du triangle 20.4 nous dit que chaque arête de $\mathcal{A}$ relie un sommet de $V_\mathcal{A} \cap V_{\mathcal{B}_a}$ et un sommet de $V_\mathcal{A} \cap V_{\mathcal{C}_a}$ (voir la figure 1.15). Cela prouve que $(V_\mathcal{A}, \mathcal{A})$ est biparti. On sait de plus que $V_\mathcal{A} \subseteq V_{\mathcal{B}_a} \cup V_{\mathcal{C}_a}$. Aussi pour tous sommets $b' \in V_\mathcal{A} \cap V_\mathcal{C}$ et $c' \in V_\mathcal{A} \cap V_\mathcal{B}$, les arêtes $\widehat{ab'}$ et $\widehat{ac'}$ existent et sont dans $\mathcal{C}$ et $\mathcal{B}$ respectivement. On déduit ensuite de la définition de croiser que $\widehat{b'c'} \in \mathcal{A}$. Cela prouve que le graphe biparti induit par $\mathcal{A}$ est complet.

(2) Il faut montrer qu'il existe dans $\mathcal{M}$ une classe de couleur reliant n'importe quelle paire de supersommets distincts X, Y de $\mathcal{M}$ donnée. Par définition, il existe

une suite de classes de couleur de $\mathcal{M}$, $\mathcal{B}_1, \ldots, \mathcal{B}_j$ ($j \geq 1$) telle que pour chaque i, $\mathcal{B}_i$ croise $\mathcal{B}_{i+1}$ où $\mathcal{B}_i$ relie les supersommets X_i et X_{i+1} ($X_1 = X$ et $X_{j+1} = Y$). Considérons une telle suite de longueur minimale. Si on a $j > 1$, alors $\mathcal{B}_1$ croise $\mathcal{B}_2$ par une classe de couleur $\mathcal{C}$ de $\mathcal{M}$. (1) implique alors que $\mathcal{C}$ relie X_1 et X_3. La suite $\mathcal{C}, \mathcal{B}_3, \ldots, \mathcal{B}_j$ est alors plus courte, ce qui contredit la minimalité de j, on a donc $j = 1$. Et finalement, $\mathcal{B}_1$ relie X et Y. □

Les simplexes générant $\mathcal{M}$ sont simplement des sous-graphes isomorphes au graphe complet sur les supersommets de $\mathcal{M}$. Cette remarque peut être étendue au multiplexe ne possédant qu'une seule classe de couleur et est au centre de la décomposition modulaire que nous aborderons plus loin.

Concentrons nous maintenant sur la relation couvrir. Nous savons déjà d'après sa définition que c'est une relation d'ordre ($\mathcal{A}$ couvre $\mathcal{B}$ si et seulement si $V_\mathcal{B} \subset V_\mathcal{A}$). Le théorème suivant étend cette relation aux multiplexes maximaux.

Théorème 23 *1. Si une classe de couleur $\mathcal{A}$ couvre une classe de couleur $\mathcal{B}$ d'un multiplexe maximal $\mathcal{M}$, alors elle couvre toutes les classes de $\mathcal{M}$ (et n'est donc pas dans $\mathcal{M}$). Nous dirons alors que le multiplexe $\mathcal{N}$ contenant $\mathcal{A}$* couvre *$\mathcal{M}$.*

2. Si deux multiplexes se touchent*, c'est-à-dire $V_\mathcal{M} \cap V_\mathcal{N} \neq \emptyset$, alors l'un couvre l'autre.*

3. Un multiplexe maximal $\mathcal{M}$ couvre un multiplexe maximal $\mathcal{N}$ si et seulement si $V_\mathcal{N} \subseteq V_\mathcal{M}$.

4. La relation couvrir sur les multiplexes maximaux est un ordre forêt, et même un ordre d'arbre quand le graphe est connexe.

Preuve. (1) Si une classe de couleur $\mathcal{C}$ croise $\mathcal{B}$, alors elle touche $\mathcal{A}$. Comme $\mathcal{A}$ couvre $\mathcal{B}$, $\mathcal{C}$ ne peut pas couvrir $\mathcal{A}$. Si $\mathcal{C}$ croise $\mathcal{A}$ alors on déduit facilement du lemme du triangle 20 que $\mathcal{C}$ couvre $\mathcal{B}$. C'est une contradiction, et le seul cas possible est $\mathcal{A}$ couvre $\mathcal{C}$. En répétant ce raisonnement, on trouve que $\mathcal{A}$ couvre toutes les classes de $\mathcal{M}$.

(2) Si $V_\mathcal{M} \cap V_\mathcal{N} \neq \emptyset$, alors il existe deux classes de couleur, une dans $\mathcal{M}$ et une dans $\mathcal{N}$ se touchant. Comme $\mathcal{M} \neq \mathcal{N}$, l'une doit couvrir l'autre et (1) permet de conclure.

(3) se déduit de (1) et (2).

(4) Soit $\mathcal{M}$ un multiplexe maximal d'un graphe $G = (V, \mathcal{E})$. Considérons l'ensemble S de tous les multiplexes maximaux couvrant $\mathcal{M}$. Comme $V_\mathcal{M} \subseteq \bigcap_{\mathcal{N} \in S} V_\mathcal{N}$ et $V_\mathcal{M} \neq \emptyset$, tous ces multiplexes maximaux se touchent les uns les autres. (2) implique donc que S est totalement ordonné par la relation couvrir. Cela prouve que la relation couvrir est un ordre forêt dans le cas d'un graphe G quelconque.

Dans le cas où G est connexe, considérons un multiplexe maximal pour la relation couvrir. Soit $a \in V_\mathcal{M}$ et $\widehat{ab} \in \mathcal{E}$. $\mathcal{M}$ touche le multiplexe contenant $\widehat{ab}$ et donc le couvre puisqu'il est maximal, d'où $b \in V_\mathcal{M}$. Comme G est connexe, on a

$V \subseteq V_{\mathcal{M}}$ et $\mathcal{M}$ est l'unique maximum de la relation couvrir qui est donc un ordre d'arbre de racine $\mathcal{M}$. □

Nous en savons maintenant assez pour en déduire des algorithmes parallèles efficaces une fois le calcul des classes d'implication effectué.

Algorithme d'orientation transitive

De manière surprenante, il est possible de calculer une orientation transitive correspondant à l'orientation de simplexes maximaux spécifiques sans calculer ni ces simplexes, ni les multiplexes maximaux. Les simplexes choisis sont ceux obtenus en prenant dans chaque supersommet le sommet de numéro minimal. Chaque simplexe est orienté selon l'ordre total induit par les numéros de ses sommets. Il suffit pour cela de calculer l'union des arêtes de ces simplexes (cela fait une arête par classe de couleur). Un exemple est illustré par la figure 1.18. Voir l'algorithme 1.14.

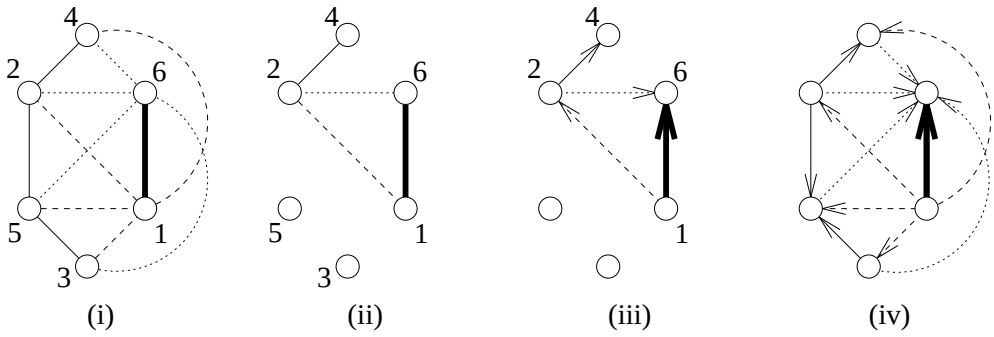


FIG. 1.18 – *(i) Le graphe de comparabilité de la figure 1.14 avec des sommets numérotés. (ii) L'union des simplexes maximaux implicitement choisis. (iii) L'union des simplexes orientés. (iv) L'orientation correspondante du graphe.*

Théorème 24 *L'algorithme 1.14 calcule une orientation d'un graphe de comparabilité dont les classes d'implication et de couleur sont données. Il s'exécute dans le modèle* EREW PRAM *en temps* $O(\log m)$ *avec* m *processeurs.*

L'exactitude de l'algorithme découle des théorèmes 22 et 19.2.

Calcul des multiplexes maximaux

Nous pourrions calculer les multiplexes maximaux de la même façon que les classes d'implication en utilisant le théorème 21, mais on peut obtenir une meilleure complexité en travaillant sur les simplexes grâce à la propriété suivante.

Algorithme 1.14 Orientation d'un graphe de comparabilité

Données : Un graphe de comparabilité, ses classes de couleur et d'implication, donnés par la liste de ses arcs et leurs numéros de classes.

Résultat : Une orientation transitive du graphe.

Etape 1 *Sélection d'un arc dans chaque classe.*

Trier les arcs par classe de couleur.

Trier les arcs lexicographiquement à l'intérieur de chaque sous-bloc d'arcs de même couleur.

Pour tout *arc ab de classe de couleur numéro c et de classe d'implication numéro i* **effectuer**

Si ab est le premier arc de couleur c dans la liste triée, alors poser $I[c] \leftarrow i$.

Etape 2 *Orientation du graphe entier.*

Pour tout *arc ab de classe de couleur numéro c et de classe d'implication numéro i* **effectuer**

Si $I[c] = i$ **Alors**

Marquer ab. {*Cette lecture concurrente peut être effectuée par* $\log m$ *lectures exclusives.*}

{*Les arcs marqués forment une orientation transitive.*}

Lemme 25 *Soit* $G = (V, \mathcal{E})$ *un graphe non orienté. Soit* $(V, \mathcal{S})$ *une union de simplexes maximaux de* G *tels qu'ils engendrent des multiplexes tous différents. Alors les simplexes maximaux sont les composantes bi-connexes de* $(V, \mathcal{S})$.

Un sous-graphe est bi-connexe s'il est connexe et encore connexe après lui avoir retiré un sommet quelconque. De manière équivalente, un sous-graphe est bi-connexe si pour toute paire d'arêtes distinctes, il existe un cycle simple (qui ne passe qu'une fois par sommet) les contenant toutes les deux.

Remarquons que l'algorithme 1.14 calcule un tel graphe $(V, \mathcal{S})$ où tous les multiplexes maximaux sont représentés.

Preuve. Chaque simplexe est bi-connexe puisque c'est un graphe complet. Supposons par contradiction qu'il existe une composante bi-connexe contenant plusieurs simplexes maximaux. Deux simplexes maximaux ne peuvent pas avoir deux sommets en commun car l'arête correspondante appartiendrait à deux multiplexes maximaux différents. Il doit donc exister un cycle simple avec des arêtes dans des simplexes différents. Comme les simplexes sont des graphes complets, il existe un cycle simple $a_1, \ldots, a_j$ dont les arêtes sont toutes dans des simplexes différents et donc dans des multiplexes maximaux différents. Considérons maintenant ce cycle en tant que cycle de G. Comme $\widehat{a_1a_2}$ et $\widehat{a_1a_j}$ sont dans des multiplexes maximaux différents, les classes de couleur de $\widehat{a_1a_2}$ et de $\widehat{a_1a_j}$ ne peuvent pas se croiser. L'arête $\widehat{a_2a_j}$ qui existe forcément, est de même couleur que $\widehat{a_1a_2}$ ou $\widehat{a_1a_j}$

et le cycle $a_2, \ldots, a_j$ a encore toutes ses arêtes dans des multiplexes maximaux distincts. En itérant le procédé, on finit par trouver un cycle a_{j-2}, a_{j-1}, a_j formant un triangle tricolore, ce qui contredit le fait que $a_{j-2}\widehat{a_{j-1}}$ et $a_{j-1}\widehat{a_j}$ sont dans des multiplexes maximaux différents. □

L'algorithme est simple : calculer un tel $(V, \mathcal{S})$ comme dans l'algorithme 1.14 et trouver ses composantes bi-connexes. Les détails sont donnés par l'algorithme 1.15.

Algorithme 1.15 Calcul des multiplexes et du nombre d'orientations transitives

Données : Un graphe non orienté *quelconque* et ses classes de couleur donnés par la liste de ses arcs et leur numéro de classes.

Résultat : Le numéro du multiplexe maximal contenant chaque classe de couleur.

Etape 1 *Calcul d'une union de simplexes maximaux.*

Trier les arcs par classe de couleur.
Trier les arcs lexicographiquement à l'intérieur de chaque sous-bloc d'arcs de même couleur.
Pour tout *arc ab de classe de couleur numéro c* **effectuer**
Si ab est le premier arc de couleur c dans la liste triée, alors poser $E[c] \leftarrow \widehat{ab}$.
Calculer les composantes bi-connexes du graphe (V, E).
Pour tout *arête* $\widehat{ab} \in E$ *de classe de couleur numéro c* **effectuer**
Identifier le numéro de multiplexe maximal de la classe de couleur c à celui de la composante bi-connexe de $\widehat{ab}$.

Etape 2 *Calcul du nombre d'orientations transitives.*

Si *le graphe n'est pas de comparabilité* **Alors**
il a 0 orientation transitive
Sinon
Trier les arêtes de la liste E selon leur numéro de multiplexe maximal.
Calculer avec une somme préfixée le nombre $N(m)$ d'arêtes dans le simplexe maximal générant chaque multiplexe de numéro m.
Le nombre de sommets du simplexe générant le multiplexe de numéro m est donné par $V(m) := \dfrac{1+\sqrt{8N(m)+1}}{2}$. {*On a* $N(m) = V(m)\,(V(m)-1)\,/2$ *puisqu'un simplexe est un graphe complet.*}
Calculer avec un produit préfixé le nombre d'orientations transitives qui est $\prod_m V(m)!$ d'après le théorème 19.3.

Les composantes bi-connexes d'un graphe de p sommets et q arêtes peuvent être calculées [80] dans le modèle CRCW PRAM en temps $O(\log p)$ avec $p + q$

processeurs. On en déduit le résultat suivant.

Théorème 26 *L'algorithme 1.15 calcule les multiplexes maximaux et le nombre d'orientations transitives de n'importe quel graphe dont les classes de couleur sont données. Il s'exécute en temps $O(\log n)$ avec $n+m$ processeurs dans le modèle* CRCW PRAM.

L'exactitude de l'algorithme découle du lemme 25.

Nous allons maintenant faire le lien entre les multiplexes et la décomposition modulaire ce qui nous permettra de produire un algorithme de décomposition modulaire grâce à l'algorithme de calcul des multiplexes maximaux. Cet algorithme n'étant pas optimal (il est possible de calculer les multiplexes maximaux à partir de la décomposition modulaire, ce qui est un problème linéaire en séquentiel), l'algorithme de décomposition modulaire proposé n'est pas non plus optimal, mais le résultat est important d'un point de vue théorique puisqu'il montre comment calculer la décomposition modulaire à partir des multiplexes maximaux.

Décomposition modulaire

Soit $G=(V,\mathcal{E}), G_1=(V_1,\mathcal{E}_1),\ldots,G_k=(V_k,\mathcal{E}_k)$ des graphes non orientés d'ensembles de sommets disjoints. Le *graphe composé* $G\binom{a_1}{G_1},\ldots,\binom{a_k}{G_k}=(V',\mathcal{E}')$ où $a_1,\ldots,a_k$ sont des sommets distincts de G, est le graphe obtenu en remplaçant dans G chaque a_i par G_i (voir la figure 1.19). Plus formellement :

$$V' = \big(V-\{a_1,\ldots,a_k\}\big)\cup V_1\cup\cdots\cup V_k$$
$$\text{et}\qquad \mathcal{E}' = \mathcal{E}_1\cup\cdots\cup\mathcal{E}_k\cup\bigcup_{i\neq j}\{ab\ a\in V_i, b\in V_j \mid a_ia_j\in\mathcal{E}\}.$$

Les arêtes dans $\bigcup_{i\neq j}\{ab\ a\in V_i, b\in V_j \mid a_ia_j\in\mathcal{E}\}$ sont dites arêtes *internes* de la composition. La composition est *propre* s'il existe i tel que $1<|V_i|<|V|$.

Un graphe $G'=(V',\mathcal{E}')$ est *décomposable* si et seulement s'il peut être obtenu par composition propre. Dans le cas contraire, G est dit *premier*. Un sous-ensemble M de V' est un *module* (ou encore est dit *homogène*) si et seulement si pour tout $a\in V'-M$, a est relié soit à tous les sommets de M soit à aucun. Un module est *propre* quand $1<|M|<|V'|$, il est dit *trivial* dans le cas contraire.

Un graphe $G'=(V',\mathcal{E}')$ est décomposable si et seulement s'il contient un module propre, on a alors $G=H\binom{a}{G_M}$ où G_M est le sous-graphe de G induit par M et H est obtenu à partir de G en remplaçant M par un unique sommet.

Théorème 27 (Gallai [34]) *Pour tout graphe $G=(V,\mathcal{E})$, un des trois cas suivants est vérifié :*

1. *$G=H\binom{a_1}{G_1},\ldots,\binom{a_k}{G_k}$, où H est un graphe indépendant (sans arêtes). G est obtenu par composition* parallèle.

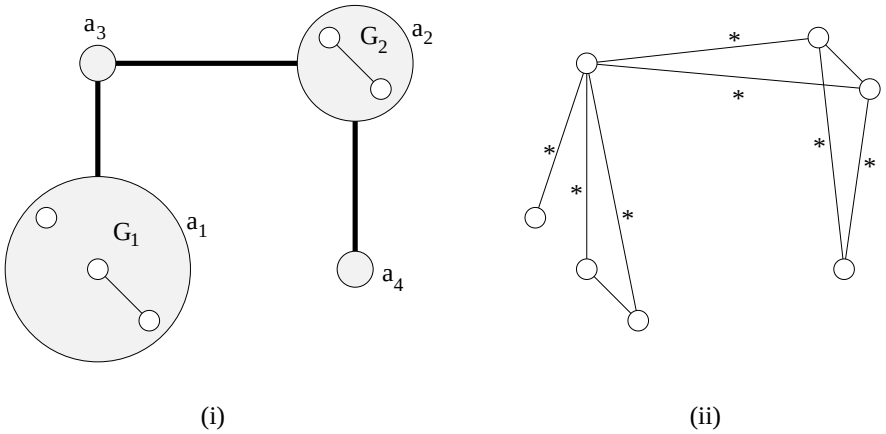


FIG. 1.19 – *(i) Un graphe G d'ensemble de sommets $\{a_1, a_2, a_3, a_4\}$. Un graphe G_1 (respectivement G_2) est représenté à l'intérieur de a_1 (respectivement a_2). (ii) Le graphe composé $G(^{a_1}_{G_1}, ^{a_2}_{G_2})$, Les arêtes marquées avec une $*$ sont internes.*

2. $G = H(^{a_1}_{G_1}, \ldots, ^{a_k}_{G_k})$, *où H est un graphe complet. G est obtenu par composition* série.
3. $G = H(^{a_1}_{G_1}, \ldots, ^{a_k}_{G_k})$, *où H est un graphe premier (unique). G est obtenu par composition de* type premier.

Ces trois cas mutuellement exclusifs permettent de représenter un graphe quelconque sous forme d'un arbre T. La racine de T est V, ses feuilles sont les sommets de G et les fils d'un nœud interne sont les ensembles de sommets des graphes G_i de la composition (parallèle, série ou de type premier) du graphe associé au nœud. L'arbre est unique si H est pris le plus grand possible dans les cas 1 et 2, il est alors appelé *arbre canonique de décomposition*. Un nœud interne de l'arbre est étiqueté par S (respectivement P) si c'est un nœud série (respectivement parallèle), et par le graphe H si c'est un nœud premier. Voir la figure 1.20.

Un arbre de décomposition donne une représentation de tous les modules dans le sens suivant : tout module du graphe correspond soit à un nœud de l'arbre, soit à l'union des sommets d'une partie des fils d'un nœud *dégénéré*, c'est-à-dire série ou parallèle. Les nœuds de l'arbre canonique correspondent aux *modules forts*, c'est-à-dire les modules n'intersectant strictement aucun autre module (ou encore ceux qui sont toujours comparables par inclusion avec les autres modules).

Multiplexes maximaux et décomposition modulaire

Nous pouvons maintenant expliquer le lien entre les multiplexes maximaux et la décomposition modulaire dans le théorème suivant. Le point 1 est le seul apparaissant dans [37].

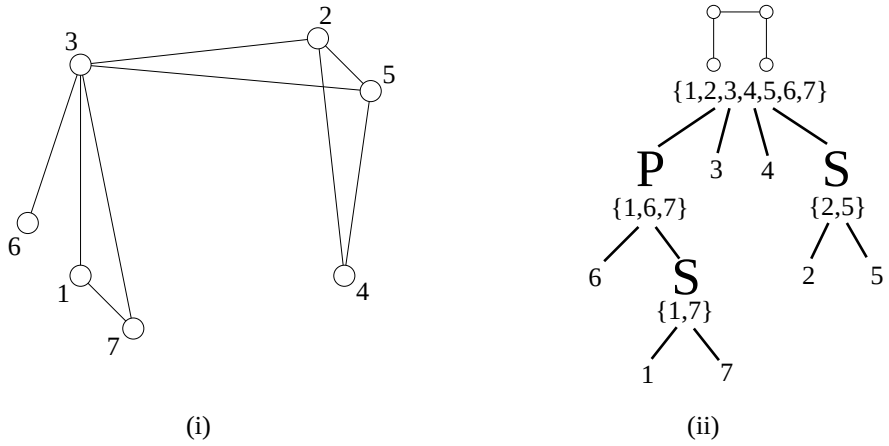


FIG. 1.20 – *(i) Le graphe de la figure 1.19. (ii) Son arbre de décomposition canonique.*

Théorème 28 *Soit $G = (V, \mathcal{E})$ un graphe non orienté.*

1. *(GOLUMBIC [37]) Pour toute classe de couleur $\mathcal{A}$, $V_{\mathcal{A}}$ est un module.*
2. *Si M un module contenant une arête $\widehat{ab}$ $(a, b \in M)$ de classe de couleur $\mathcal{A}$, alors on a $V_{\mathcal{A}} \subseteq M$.*
3. *Pour tout multiplexe maximal $\mathcal{M}$, $V_{\mathcal{M}}$ est un module fort.*
4. *Si M un module fort contenant une arête $\widehat{ab}$ $(a, b \in M)$ de multiplexe maximal $\mathcal{M}$, alors on a $V_{\mathcal{M}} \subseteq M$.*

Preuve. (1) se déduit directement du lemme du triangle 20.2.

(2) Soit c tel que $ab \sim ac$. Comme M est un module contenant a et b, il doit aussi contenir c qui est relié à a mais pas à b. En répétant suffisamment ce procédé, on finit par conclure $V_{\mathcal{A}} \subseteq M$.

(3) Supposons qu'il existe $c \in V - V_{\mathcal{M}}$ adjacent à un sommet $a \in V_{\mathcal{M}}$. Soit $\widehat{ab} \in \mathcal{M}$. Comme $c \in V - V_{\mathcal{M}}$, la classe de couleur $\mathcal{B}$ contenant $\widehat{ac}$ n'est pas dans $\mathcal{M}$. $\widehat{bc}$ existe puisque $ac \not\sim ab$. Comme $\mathcal{B}$ ne croise pas la classe de couleur $\mathcal{A}$ contenant $\widehat{ab}$ et $V_{\mathcal{B}} \not\subseteq V_{\mathcal{A}}$, $\mathcal{B}$ couvre $\mathcal{A}$ et $\widehat{bc} \in \mathcal{B}$. En itérant ce procédé, on finit par prouver que toutes les arêtes $\widehat{dc}$ existent (et sont dans $\mathcal{B}$). $V_{\mathcal{M}}$ est donc un module.

Prouvons maintenant que $V_{\mathcal{M}}$ est un module fort. Supposons qu'il existe un module N intersectant strictement $V_{\mathcal{M}}$. Comme $(V_{\mathcal{M}}, \mathcal{M})$ est connexe, il existe $a \in V_{\mathcal{M}} - N$ et $b \in N \cap V_{\mathcal{M}}$ tel que $\widehat{ab} \in \mathcal{M}$. Comme N est un module, $\widehat{ac}$ existe pour tout $c \in N$. Soit $d \in N - V_{\mathcal{M}}$ et $\mathcal{B}$ la classe de couleur contenant $\widehat{ad}$. On vient de prouver que l'on a alors $V_{\mathcal{M}} \subseteq V_{\mathcal{B}}$. Mais (2) implique alors la contradiction $V_{\mathcal{M}} \subseteq V_{\mathcal{B}} \subseteq N$.

(4) Soit $\mathcal{A}$ la classe de couleur contenant $\widehat{ab}$. on sait d'après (2) que $V_{\mathcal{A}} \subseteq M$. Soit $\mathcal{B}$ une classe de couleur croisant $\mathcal{A}$. $V_{\mathcal{B}}$ est un module intersectant strictement

$V_{\mathcal{A}}$. Comme $\mathcal{M}$ est un module fort, il doit contenir $V_{\mathcal{B}}$. En itérant ce procédé, on finit par conclure $V_{\mathcal{M}} \subseteq M$. □

Nous voyons maintenant que le théorème 22 est simplement un théorème de décomposition pour les nœuds séries. Nous pouvons généraliser cela comme suit (voir aussi la figure 1.21).

Théorème 29 *L'arbre de décomposition canonique d'un graphe non orienté $G = (V, \mathcal{E})$ vérifie les propriétés suivantes.*

1. *Tout nœud interne non parallèle correspond à un module fort de la forme $V_{\mathcal{M}}$ où $\mathcal{M}$ est un multiplexe maximal. De plus, $\mathcal{M}$ est l'ensemble des arêtes internes de la composition.*
2. *Un nœud correspondant à un module $\mathcal{M}$ est un ancêtre d'un nœud correspondant à un module $\mathcal{N}$ si et seulement si $\mathcal{M}$ couvre $\mathcal{N}$, ou de manière équivalente $V_{\mathcal{N}} \subseteq V_{\mathcal{M}}$.*

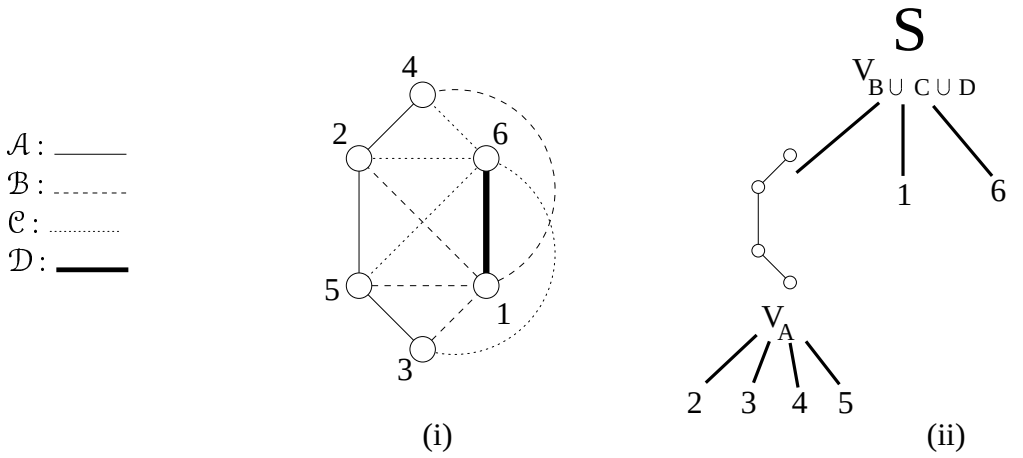


FIG. 1.21 – *(i) Le graphe de la figure 1.14. Ses multiplexes maximaux sont $\mathcal{A}$ et $\mathcal{B} \cup \mathcal{C} \cup \mathcal{D}$. (ii) Son arbre de décomposition canonique.*

Preuve. (1) Considérons la composition $G = H\left(\begin{smallmatrix} a_1 \\ G_1 \end{smallmatrix}, \ldots, \begin{smallmatrix} a_k \\ G_k \end{smallmatrix}\right)$. Il est clair que les classes de couleur des sous-graphes $G_1, \ldots, G_k$ sont les mêmes que dans G. Il suffit donc de faire la preuve pour la racine de l'arbre.

Remarquons que dans une composition non parallèle, l'ensemble $\mathcal{F}$ des arêtes internes de la composition induisent un sous-graphe connexe, ce qui implique $V_{\mathcal{F}} = V$. Considérons tout d'abord le cas où la racine est un nœud série. On déduit du théorème 22 et de la « maximalité » des nœuds séries que $\mathcal{F}$ est un multiplexe maximal.

Considérons maintenant un graphe $G = H\left(\begin{smallmatrix} a_1 \\ G_1 \end{smallmatrix}, \ldots, \begin{smallmatrix} a_k \\ G_k \end{smallmatrix}\right)$ résultant d'une composition où $H = (W, \mathcal{F}')$ est premier. Soit ab un arc interne. Si ab force directement un arc ac, alors b est relié à a mais pas à c et ac doit donc être interne aussi. Ceci

prouve que toutes les arêtes de même couleur que $\widehat{ab}$ sont internes. Supposons par l'absurde que $\mathcal{F}$ contient plusieurs classes de couleur. Comme $\mathcal{F}$ induit un sous-graphe connexe, il doit contenir deux classes de couleur $\mathcal{A}, \mathcal{B}$ se touchant. On peut supposer sans perte de généralité que $\mathcal{A}$ croise $\mathcal{B}$ ou $\mathcal{A}$ couvre $\mathcal{B}$. Dans les deux cas, le théorème 28.1 implique que $V_{\mathcal{A}}$ est un module propre de G. On peut alors facilement montrer que $M = \{a_i \mid V_i \text{ contient un sommet de } V_{\mathcal{A}}\} \bigcup V_{\mathcal{A}} \cap W$ est un module de H. De plus M contient au moins deux sommets puisque $\mathcal{A}$ contient des arêtes internes. M est donc un module propre, ce qui contredit la primalité de H.

(2) est évident. □

Ce résultat conduit naturellement à concevoir un algorithme de décomposition modulaire.

Algorithme parallèle de décomposition modulaire

L'algorithme suivant de décomposition modulaire est uniquement basé sur le calcul des multiplexes maximaux. Comme les compositions parallèles ne possèdent pas d'arêtes internes, ils ne sont associés à aucun multiplexe maximal. On commence donc par calculer le reste de l'arbre et on insère ensuite les nœuds premiers que l'on calcule grâce à un algorithme de composantes connexes. Remarquons qu'un graphe de n sommets a au plus $n-1$ multiplexes maximaux puisque son arbre de décomposition canonique a n feuilles et donc au plus $n-1$ nœuds internes. Voir l'algorithme 1.16.

Théorème 30 *L'algorithme 1.16 calcule l'arbre de décomposition canonique d'un graphe non orienté dont les multiplexes maximaux sont donnés. Il s'exécute dans le modèle* CRCW PRAM *en temps* $O(\log n)$ *avec* $n+m$ *processeurs.*

L'exactitude de l'algorithme découle des théorèmes 27 et 29. Sa complexité découle de celle du tri et du calcul des composantes connexes. Remarquons que le graphe associé à un multiplexe $\mathcal{M}$ a moins de sommets que $V_{\mathcal{M}}$ et moins d'arêtes que $\mathcal{M}$. L'ensemble de ces graphes a donc bien une taille en $O(n+m)$.

Résumé

L'algorithme 1.13 calcule les classes de couleur et d'implication d'un graphe non orienté quelconque et détermine s'il est de comparabilité. L'algorithme 1.14 calcule une orientation transitive d'un graphe dont les classes d'implication sont données. L'algorithme 1.15 calcule les multiplexes maximaux d'un graphe dont les classes de couleur sont données. L'algorithme 1.16 calcule l'arbre de décomposition canonique d'un graphe dont le complémentaire et les multiplexes maximaux des deux graphes sont donnés.

Algorithme 1.16 Décomposition modulaire

Données : Un graphe non orienté G et ses multiplexes maximaux.

Résultat : L'arbre canonique de décomposition de G.

Etape 1 *Calcul des nœuds non parallèles de l'arbre.*

Pour tout arc ab de multiplexe maximal de numéro m, trier lexicographiquement les triplets (a, m, b).
Trouver grâce à un calcul préfixé la liste L des couples (a, m) apparaissant au début d'un triplet.
Trier L anti-lexicographiquement pour obtenir $V_{\mathcal{M}}$ pour chaque multiplexe maximal de numéro m.

Etape 2 *Calcul du père de chaque nœud.*

Pour chaque multiplexe maximal $\mathcal{M}$ de G, trouver $|V_{\mathcal{M}}|$ à l'aide de calculs préfixés. Lire dans la liste L triée lexicographiquement les listes triées $N(a)$ de tous les nœuds contenant chaque sommet a. **Pour tout** *nœud* $\mathcal{M}$ **effectuer**

Prendre un arc ab dans le multiplexe de G ou de $\overline{G}$ correspondant. {*Comme* $\widehat{ab}$ *est une arête interne de la composition correspondante, un descendant de* $\mathcal{M}$ *ne peut pas contenir à la fois* a *et* b*. La liste des ancêtres* $A(\mathcal{M})$ *est donc donnée par* $N(a) \cap N(b) - \{\mathcal{M}\}$.}
Calculer $A(\mathcal{M})$ en fusionnant $N(a)$ et $N(b)$.
Calculer avec un calcul préfixé le père de $\mathcal{M}$ qui est l'élément de $A(\mathcal{M})$ de cardinal minimal.

Pour toute feuille a, le père de a est l'élément de $N(a)$ de cardinal minimal.

Etape 3 *Calcul des nœuds parallèles.*

Marquer dans chaque $V_{\mathcal{M}}$ un sommet de chacun de ses fils, de manière à obtenir le graphe $H_{\mathcal{M}} = (V^*_{\mathcal{M}}, \mathcal{M} \cap V^*_{\mathcal{M}} \times V^*_{\mathcal{M}})$ étiquetant le nœud $\mathcal{M}$, où $V^*_{\mathcal{M}}$ est l'ensemble des sommets marqués de $V_{\mathcal{M}}$.
Calculer les composantes connexes de chacun des $H_{\mathcal{M}}$ et introduire un nœud parallèle chaque fois qu'il y en a plusieurs. Les anciens fils de $\mathcal{M}$ sont rattachés à la composante contenant leur sommet marqué.
Les nœuds séries sont ceux correspondant à une composante connexe qui est un graphe complet, ce qui peut se détecter en en comptant le nombre d'arcs.

En combinant ces résultats, on obtient un algorithme d'orientation transitive et un algorithme de décomposition modulaire. Ils s'exécutent tous les deux en temps $O(\log n)$ dans le modèle CRCW PRAM avec δm processeurs.

Leur complexité rend ces deux algorithmes peu efficaces par rapport aux algorithmes séquentiels qui sont linéaires [58, 57, 15] ; ils peuvent toutefois être utiles pour traiter des graphes qui ne tiennent pas dans la mémoire d'un ordinateur séquentiel. Rappelons que l'algorithme de DALHAUS [18] effectue la décomposition

modulaire en parallèle avec un nombre de processeurs linéaire. Cela laisse supposer qu'il doit exister un algorithme avec une complexité similaire pour l'orientation transitive en parallèle. Cependant, les algorithmes intermédiaires ont des conséquences théoriques intéressantes.

Remarquons que les classes de couleur se déduisent facilement de l'arbre de décomposition canonique, en effet toute arête $\widehat{ab}$ est une arête interne de la composition correspondant au plus petit ancêtre commun de a et b (les classes de couleur qui ne sont pas des multiplexes maximaux se déduisent facilement de la structure des nœuds séries). Les algorithmes 1.16 et 1.15 permettent donc d'affirmer qu'il est aussi difficile de trouver les classes de couleur que de calculer la décomposition modulaire.

De plus, étant donnée la décomposition modulaire d'un graphe (ou de manière équivalente ses classes de couleur), une orientation transitive $\mathcal{F}$ en donne les classes d'implication qui sont de la forme $\mathcal{A} \cap \mathcal{F}$ pour toute classe de couleur $\mathcal{A}$. L'algorithme 1.14 permet donc d'affirmer qu'il est alors aussi difficile de calculer les classes d'implication qu'une orientation transitive.

1.5 La barrière de la fermeture transitive

Cette section est consacrée à quelques problèmes ouverts, le plus important d'entre eux en ce qui concerne l'algorithmique parallèle des graphes est sans doute celui de la barrière de la fermeture transitive. Derrière les solutions parallèles particulières que nous avons vu dans ce chapitre se cache un problème profond de l'algorithmique parallèle des graphes. De nombreux algorithmes parallèles de graphes orientés font une hypothèse sur la forme de l'entrée vis à vis de la fermeture transitive. L'entrée est par exemple supposée réduite transitivement dans le cas des ordres N-free. L'algorithme de reconnaissance des ordres d'intervalles présenté dans[5] a besoin d'une entrée fermée transitivement. En ce qui concerne les ordres de dimension fixée, les permutations forment une représentation de la fermeture transitive. Ces hypothèse permettent d'éviter de se confronter à certains problèmes qui sont simples en séquentiel, mais que l'on ne sait résoudre en parallèle que par le calcul de la fermeture transitive.

Les problèmes confrontés à la barrière de la fermeture transitive

Karp et Ramachadran [47] donnent la liste suivante de problèmes de graphes orientés que l'on peut résoudre en temps polylogarithmique dans le modèle pram grâce au produit de matrices carrées.

1. Calculer les composantes fortement connexes d'un graphe orienté.
2. Déterminer si un graphe orienté est exempt de circuits.
3. Construire un tri topologique d'un graphe orienté sans circuit.

4. Construire un arbre enraciné de racine un sommet donné d'un graphe orienté et contenant tous les sommets atteignables depuis ce sommet.
5. Construire un arbre de recherche en largeur à partir d'un sommet donné d'un graphe orienté.
6. Construire les plus courts chemin depuis un sommet donné vers tous les autres sommets dans un graphe orienté où les arcs sont étiquetés par des poids positifs ou nuls.

Ces problèmes sont des briques élémentaires fréquemment utilisées en algorithmique des graphes orientés. Ils sont tous dans la classe NC des problèmes dont la solution peut être calculée par un algorithme PRAM en temps polylogarithmique avec un nombre polynômial de processeurs (en fonction de la taille du problème). Toutes les solutions connues à ce jour utilisent la fermeture transitive ou plus généralement le produit de matrices carrées (un nombre logarithmique de produits de matrices carrées booléennes permet de calculer la fermeture transitive d'un graphe à partir de sa matrice d'adjacence). Ces solutions sont très loin d'être optimales car il existe des algorithmes séquentiels linéaires permettant de résoudre ces problèmes[6]. Notre incapacité à trouver des algorithmes parallèles pour résoudre ces problèmes évitant le produit de matrice ou la fermeture transitive est souvent appelée *barrière de la fermeture transitive* («*transitive closure bottleneck* » en anglais).

Rappelons les différentes définitions. Un sous-ensemble A de sommets d'un graphe orienté est *fortement connexe* si et seulement s'il existe un chemin orienté de tout sommet de A vers tout autre sommet de A ; les composantes fortement connexes sont les sous-ensembles maximaux de sommets vérifiant cette propriété. Un *tri topologique* d'un graphe orienté sans circuit ou extension linéaire de l'ordre associé (obtenu en fermant transitivement) est une permutation $u_1, \ldots, u_n$ des sommets telle que tout arc $u_i u_j$ vérifie $i < j$.

La réponse aux problèmes 1 et 2 se calcule facilement à partir de la fermeture transitive du graphe orienté dont il est question : deux sommets u et v sont dans la même composante fortement connexe si uv et vu sont des arcs de la fermeture transitive ; un graphe orienté est sans circuit si et seulement si aucun arc de sa fermeture transitive ne relie un sommet à lui-même. Le problème 6 peut être résolu [47] avec $\log n$ élévations au carré de la matrice des poids des arcs où les opérations du produit de matrice sont min et $+$. Le problème 5 est un cas particulier du problème 6 et offre une solution au problème 4. KARP et RAMACHADRAN proposent la solution suivante au problème 3 : calculer la longueur d'un plus long chemin menant à chaque sommet (et partant de n'importe quel sommet) par $\log n$ élévations au carré de la matrice d'adjacence du graphe orienté sans circuit, puis trier les sommets selon cette valeur associée.

Nous pouvons poursuivre un peu l'étude de KARP et RAMACHADRAN. La valeur dont il est question juste au dessus s'appelle en théorie des ordres la *hauteur*

6. Pour être précis, le problème 6 peut être résolu en séquentiel en temps $O(n \log n + m)$.

du sommet considéré. Un plus long chemin menant à ce sommet partira forcément d'une source (c'est-à-dire d'un sommet sans prédécesseur). En triant les sommets par hauteur (en ordonnant les sommets de même hauteur de manière arbitraire), on obtient bien une extension linéaire. On peut donc rajouter le problème suivant à la liste de KARP et RAMACHADRAN :

7. Calculer la hauteur des sommets d'un graphe orienté sans circuit.

Remarquons qu'il existe un autre moyen de calculer une extension linéaire d'un graphe orienté sans circuit : trier les sommets suivant leur degré dans la fermeture transitive. En effet, si le graphe contient un arc uv, alors u a au moins un successeur de plus que v dans la fermeture transitive (le sommet v lui-même) et le degré de u dans la fermeture transitive est strictement supérieur à celui de v (il faut trier les sommets par degré décroissant). Pour mieux cerner le problème de la barrière transitive, je ne m'interesserai par la suite qu'aux problèmes suivants qui peuvent tous être résolus grâce à un calcul de fermeture transitive :

(i). Calculer les composantes fortement connexes d'un graphe orienté.

(ii). Déterminer si un graphe orienté est exempt de circuits.

(iii). Construire un tri topologique d'un graphe orienté sans circuit.

(iv). Calculer l'ensemble des sommets atteignables depuis un sommet donné d'un graphe orienté (nous appelleront cet ensemble l'ensemble *atteignable* depuis le sommet considéré).

J'ai remplacé les problèmes 4, 5 et 6 par un problème plus simple : calculer les successeurs dans la fermeture transitive d'un sommet donné. Le calcul de la fermeture transitive consiste à se poser ce problème pour tous les sommets. En séquentiel, cela conduit à un algorithme rapide de fermeture transitive pour les graphes ayant peu d'arcs puisque sa complexité est en $O(n(n+m))$. Si $m = O(n)$ (c'est le cas pour un graphe de degré borné par exemple), alors cet algorithme prend un temps $O(n^2)$. Cette approche est donc intéressante et nous amène à soulever la question en parallèle :

> Existe-t-il un algorithme parallèle de fermeture transitive dont la complexité en travail soit à un facteur polylogarithmique de $O(n(n+m))$?

A ma connaissance, la seule réponse allant dans le sens de cette question que l'on puisse trouver dans la littérature est celle de ULLMAN et YANNAKAKIS [82] qui proposent un algorithme probabiliste pour le problème (iv) qui tourne dans le modèle PRAM grosso modo en temps $O(\sqrt{n})$ avec $O(m)$ processeurs. Ils proposent aussi un algorithme probabiliste de fermeture transitive en temps $O(\sqrt{n})$ avec $O(m\sqrt{n})$ processeurs. Le « grosso modo » signifie que les bornes de complexité de ces deux algorithmes sont données à un facteur polylogarithmique de n près. L'idée principale consiste à effectuer en parallèle plusieurs recherches en largeur jusqu'à une profondeur de $\sqrt{n}$ en partant de différents sommets tirés au hasard. Ils proposent aussi une construction permettant d'obtenir des algorithmes pour les deux problèmes de complexités respectives de $O(n^\varepsilon)$ en temps avec $O(n^{1-2\varepsilon}m)$

processeurs et n^ε en temps avec $O(n^{1-\varepsilon}m)$ processeurs pourvu que $0 < \varepsilon \leq 1/2$ et $m \geq n^{2-3\varepsilon}$ (les bornes sont à nouveau à un facteur polylogarithmique près). Le résultat est très intéressant en ce qui concerne la fermeture transitive car leur algorithme présente un travail en $O(nm)$, même si c'est un résultat plutôt théorique que pratique. En revanche, l'algorithme de calcul de l'ensemble atteignable depuis un sommet donné est au mieux à un facteur $\sqrt{n}$ de l'optimal en travail. L'algorithmique probabiliste ouvre certainement des voies qui n'ont pas encore été explorées jusqu'au bout.

Réduction du problème

Il n'existe pas une classe des problèmes confrontés à la barrière de la fermeture transitive au sens où personne n'a trouvé de réduction des différents problèmes les uns dans les autres. Cette section s'inscrit cependant dans cette démarche. Une solution à certains problèmes permet en effet de résoudre d'autres problèmes.

Définition 31 *Nous dirons qu'un problème A se réduit dans un problème B pour la barrière de la fermeture transitive si la donnée d'un algorithme résolvant le problème B en temps $T(n_B, m_B)$ avec $P(n_B, m_B)$ processeurs permet de résoudre le problème A en temps $\left(T(n_A, m_A) + \frac{n_A+m_B}{P(n_A,m_A)}\right)\log^k n$ avec $P(n_A, m_A)$ processeurs pour un certain k.*

n_A, m_A désignent le nombre de sommets et le nombre d'arcs du graphe considéré pour le problème A. n_B, m_B désignent le nombre de sommets et le nombre d'arcs du graphe considéré pour le problème B. On a en tête à priori $T(n_B, m_B) = \log^l n_B$ pour un certain l et $P(n_B, m_B) = n_B + m_B$, mais cette définition permet de ne pas écarter des algorithmes d'un autre type que ceux de la classe NC comme ceux proposés dans [82] par exemple. L'idée est de résoudre un problème A en résolvant un nombre polylogarithmique de problèmes de type B sur des graphes de taille comparable.

Des composantes fortement connexes à la détection de circuit

Si un graphe orienté ne possède pas de circuit, ses composantes fortement connexes sont réduites à un seul sommet chacune. Un algorithme de calcul des composantes fortement connexes d'un graphe permet donc de déterminer avec la même complexité s'il possède un circuit.

Des composantes fortement connexes à l'ensemble atteignable depuis un sommet

Etant donné un sommet u d'un graphe orienté G, rajoutons un arc de tout autre sommet vers ce sommet. Si un sommet est atteignable depuis u dans le graphe G originel, il est alors dans la même composante fortement connexe que u dans le graphe modifié. Réciproquement, soit v un sommet dans la même composante fortement connexe que u dans le graphe modifié. Il existe par définition un chemin orienté de u vers v dans ce graphe ; considérons un plus court chemin de

u vers v. Comme les arcs que l'on a rajouté ont destination u, ce chemin ne peut emprunter un de ces arcs, ce qui signifie que v est atteignable depuis u dans G. On vient de montrer que la composante fortement connexe de u dans le nouveau graphe est exactement l'ensemble atteignable depuis u dans G.

Du tri topologique à la détection de circuit

Supposons que l'on dispose d'un algorithme calculant un tri topologique du graphe orienté sans circuit que l'on lui donne en entrée et que l'on en connaisse une borne de la complexité en temps $T(n, m)$. On peut alors le transformer en algorithme de détection de circuit prenant n'importe quel graphe orienté en entrée utilisant autant de processeurs et prenant un temps au plus $T(n, m) + 1$. Il suffit pour cela d'exécuter l'algorithme sur une entrée qui peut posséder des circuits. Si jamais on obtient une erreur à l'exécution ou si l'exécution de l'algorithme dure plus longtemps que $T(n, m)$, l'algorithme ne fonctionne pas comme il devrait, c'est donc que l'entrée possède un circuit. Si l'exécution de l'algorithme se termine dans les temps, on peut vérifier en temps constant s'il a bien calculé un tri topologique : vérifier pour chaque arc que son origine apparaît avant sa destination dans le tri topologique. Si l'algorithme a bien calculé une extension linéaire, le graphe en entrée est sans circuit et on en a même calculé un tri topologique ; dans le cas contraire, le graphe en entrée possède forcément un circuit (d'après l'exactitude de l'algorithme de calcul d'un tri topologique).

De l'ensemble atteignable depuis un sommet au tri topologique

Cette réduction est plus délicate.

L'idée est la suivante : pour calculer un tri topologique, on peut trier l'ensemble des sommets comme dans quicksort en utilisant l'algorithme de calcul des sommets atteignables depuis le pivot pour déterminer les éléments qui doivent être mis après le pivot. En utilisant ce même algorithme sur le graphe obtenu en inversant tous les arcs, on peut trouver les sommets depuis lesquels on peut atteindre le pivot. Ces sommets doivent être placés avant le pivot ; les autres sommets peuvent être placés avant ou après le pivot de manière à équilibrer au mieux la partition en deux de l'ensemble des sommets. On fait ensuite de même récursivement dans les deux sous graphes induits par les sommets placés avant le pivot d'une part et par les sommets placés après le pivot d'autre part. De même que quicksort, cet algorithme aura une profondeur de récursion en $O(\log n)$ avec une forte probabilité (c'est-à-dire que le probabilité pour que ce ne soit pas le cas est inférieure à $1/n^c$ où c est une constante multiplicative dans $O(\log n)$).

De la détection de circuit au calcul d'ensemble atteignable depuis un sommet

On peut facilement calculer l'ensemble des sommets atteignables depuis un sommet u donné d'un graphe orienté sans circuit avec n exécutions d'un algorithme de détection de circuit. Il suffit pour cela de tester pour chaque sommet v si le graphe obtenu en rajoutant l'arc vu a un circuit ou pas. Ceci n'est pas une réduction, mais le résultat est intéressant car il donne un algorithme de travail inférieur à la fermeture transitive dans le cas où $m = O(n)$ et où on aurait un algorithme de détection de circuit de travail linéaire.

La figure 1.22 illustre les différentes réductions présentées ici. Le problème de la détection de circuit apparaît comme le problème le plus simple pour lequel on bute sur la barrière de la fermeture transitive. On peut cependant considérer d'autres simplifications en supposant que l'entrée à une forme la plus spécifique possible.

Simplification du problème en spécifiant la forme de l'entrée

Si l'on se concentre sur le problème du tri topologique, l'entrée la plus spécifique que l'on puisse exiger est un graphe orienté sans circuit qui représente un ordre total, c'est-à-dire qui n'admette qu'un seul tri topologique. Même si l'on suppose que chaque sommet a degré entrant au plus deux et degré sortant au plus deux, je ne connais pas de solution en temps polylogarithmique utilisant moins de $M(n)$ processeurs, où $M(n)$ est le nombre de processeurs nécessaires pour effectuer le produit de matrices $n \times n$ en temps polylogarithmique. Je ne connais pas non plus d'algorithme probabiliste avec ces exigences en temps et en nombre de processeurs.

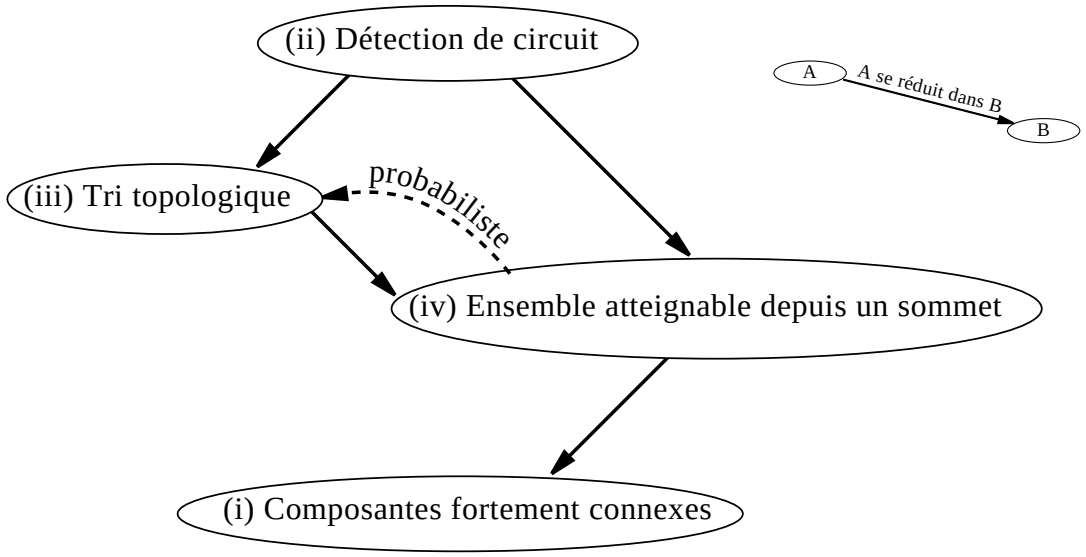


FIG. 1.22 – *Différentes réductions de problèmes posant le problème de la barrière de la fermeture transitive vers d'autres.*

Bornes inférieures

Un autre moyen de cerner le problème consiste à essayer de prouver des bornes inférieures sur la complexité de tels problèmes. Mais dans le cas de la barrière de la fermeture transitive, les arguments ne doivent pas à priori pouvoir passer au séquentiel (à moins que l'on démontre que la fermeture transitive est un problème linéaire). Cette remarque montre à quel point il peut être difficile de prouver une borne inférieure si jamais ces problèmes sont aussi difficiles que la fermeture transitive dès lors que l'on exige un temps polylogarithmique.

Comment résoudre ces problèmes sur une machine parallèle réelle ?

Il y a des graphes dont on ne pourra pas calculer un tri topologique ou les composantes fortement connexes. Si l'on considère un graphe creux (d'un nombre d'arcs de l'ordre du nombre de sommets) qui est trop volumineux pour tenir dans une machine séquentielle (il suffit qu'il possède quelques millions de sommets), on ne pourra pas calculer sa fermeture transitive si sa taille est de l'ordre n^2 (il n'existe pas de machine parallèle à un million de processeurs). Pour cela, il faudrait concevoir des algorithmes parallèles utilisant un espace mémoire linéaire.

Ce point de vue remet en cause la classe NC en tant que classe permettant de distinguer les problèmes parallélisables de ceux qui ne le sont pas. La contrainte en espace est très forte lorsque l'on traite des problèmes volumineux, et c'est l'un des principaux intérêts du parallélisme de traiter des problèmes qui sont trop volumineux pour une machine séquentielle.

Conclusion

L'approche par le parallélisme amène une formulation très synthétique de la résolution d'un problème par l'organisation de la structure de données le représentant sous une certaine forme. Dans le cas des représentation compacte, nous avons vu comment calculer l'intersection d'un ordre total avec un ordre de contiguïté. Nous avons vu comment trier la liste des arcs d'un ordre pour vérifier qu'il est N-free et en calculer un diagramme d'arcs. Nous avons enfin vu comment sélectionner un sous-ensemble d'arcs permettant d'orienter transitivement un graphe de comparabilité ou un sous-ensemble d'arcs permettant de calculer les multiplexes maximaux (ou encore les modules forts) d'un graphe quelconque. Cette approche me paraît très riche et propice à la mise en valeur des techniques essentielles utiles pour la résolution d'un problème. Le problème de la barrière de la fermeture transitive reste à mon sens un problème de fond en algorithmique parallèle des graphes. Nous avons vu cependant sur des problèmes précis qu'il est possible de contourner ce problème lorsque l'on arrive à exprimer le problème par des propriétés locales, ce n'est visiblement pas le cas pour le problème de trouver un circuit dans un graphe. Les outils de base s'avèrent être le tri, les calculs préfixés et le calcul des composantes connexes.

Chapitre 2

Modèle à gros grain cgm

Le modèle PRAM conduit à concevoir des algorithmes de grain le plus fin qui soit : un processeur par élément de l'entrée, ce qui est très loin de la réalité. Quelques modèles récents plus réalistes, BSP, C^3, et CGM [85, 43, 26], permettent de mieux prédire les performances d'algorithmes sur les machines parallèles existantes qui sont en général à gros grain. Le *grain* est le rapport de la taille de l'entrée sur le nombre de processeurs.

Le modèle CGM prend de ce point de vu le contre-pied du modèle PRAM puisqu'il fait l'hypothèse d'une mémoire locale sur chaque processeur assez importante, typiquement supérieure au nombre de processeurs. Cette hypothèse est assez forte car elle permet pour certains problèmes de trouver une solution parallèle en distribuant les données de manière adéquate, en utilisant localement le meilleur algorithme séquentiel pour ce problème, et en combinant finalement de manière assez simple les solutions locales pour trouver une solution globale. « Une fois que l'on a trouvé un algorithme CGM, il ne reste plus qu'à trouver un bon algorithme séquentiel pour l'implanter. » me disait un adepte du modèle CGM [28]. Nous verrons que cette approche ne recouvre pas tous les cas de figure et qu'il existe des algorithmes pour lesquels il est nécessaire d'effectuer des calculs vraiment globaux.

Pour simplifier, CGM permet de s'intéresser à la meilleure manière de distribuer les données dans un modèle qui fait abstraction du réseau de communications de la machine. Pour certains problèmes, cette approche seule suffit à trouver un algorithme parallèle. Mais en règle générale, il faut faire une part de calculs globaux pour se ramener à cette situation et finir par des calculs locaux. Dans ce cas, c'est l'algorithmique PRAM qui apporte la solution. En schématisant, il y a deux manières de calculer en parallèle : faire travailler plusieurs processeurs sur les mêmes données — c'est l'objet du modèle PRAM — et faire travailler séparément plusieurs processeurs sur des données différentes de manière séquentielle. Nous verrons sur quelques exemples comment le modèle CGM nous permet de combiner ces deux approches, c'est-à-dire des algorithmes PRAM et des algorithmes séquentiels, pour obtenir de bons compromis en vu d'une implantation

sur machine.

Après avoir présenté le modèle CGM nous verrons tout d'abord sur l'exemple du calcul des composantes connexes comment combiner l'algorithme PRAM pour pouvoir se ramener à des calculs locaux de composantes connexes. Nous reprendrons ensuite les trois problèmes abordés dans le chapitre consacré à des algorithmes PRAM. Nous verrons que les algorithmes PRAM basés sur les procédures de base, tels que l'algorithme de reconnaissance des ordres N-free, s'implantent assez facilement dans le modèle CGM. En revanche, le problème des ordres de dimension d et surtout sa version en termes de bases de données nous confronte directement au problème de la distribution des données. L'exemple de la reconnaissance des graphes de comparabilité nous permettra enfin d'illustrer la difficulté de traiter en parallèle une structure de données aussi irrégulière que celle associée à un graphe.

Dans ce chapitre, p désignera toujours le nombre de processeurs utilisés.

2.1 Modèle d'ordinateur à gros grain CGM

Le modèle CGM (pour «*Coarse Grained Multicomputer* »), connu aussi sous le nom de «*weak*-CREW BSP » [38] est une simplification du modèle BSP [85]. Le modèle BSP (pour «*bulk-synchronous parallel* ») est présenté par VALIANT [85] comme un modèle pont entre les langages parallèles de haut niveau (y compris le langage PRAM) et les machines parallèles. Son but est d'arriver à un analogue parallèle du modèle séquentiel de VON NEUMANN. Un programmeur écrit un programme générique pour v processeurs virtuels. L'idée de base est qu'il est possible de simuler optimalement ce programme sur une machine parallèle pourvu que le nombre de processeurs réels soit nettement inférieur à v (par exemple $v \geq p \log p$). Cette idée vient du fait que les communications prennent un temps plus long qu'un accès à la mémoire locale. D'un point de vu théorique, il est logique de compter une borne inférieure de l'ordre de $\log p$ pour accéder à une mémoire distribuée sur p unités.

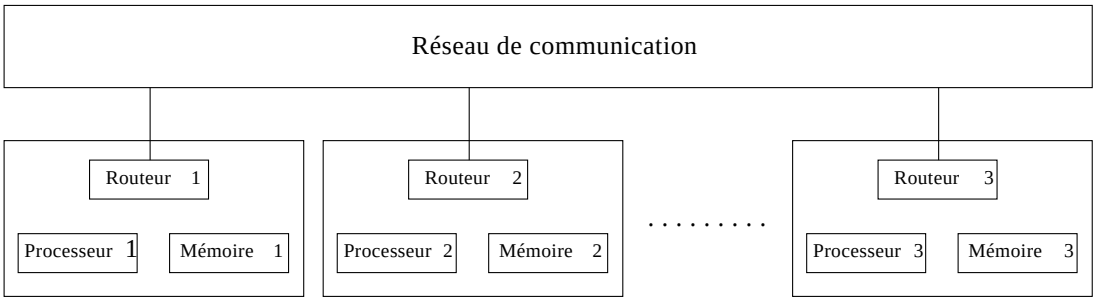


FIG. 2.1 – *Schématisation d'une machine parallèle dans les modèles* CGM *et* BSP.

Le modèle BSP permet d'exploiter cette idée grâce à des barrières de synchronisation en considérant séparément des unités de calcul et des unités de *routage* qui acheminent des messages point à point entre les unités de calcul. Les barrières de synchronisation sont effectuées périodiquement toutes les L unités de temps. Durant ce temps les unités de calcul opèrent L opérations élémentaires sur leur mémoire locale et donc sur les données qu'elles possédaient lors de la dernière synchronisation, et les unités de routage réalisent des *relations-h*, c'est-à-dire l'acheminement de messages tels que chaque unité de calcul envoie et reçoive au plus h messages. Ces communications requièrent un temps $\overline{g}h + s$ où $\overline{g}$ est le débit du routeur et s le temps de latence. Plus simplement, on borne ce temps par gh pour $g = 2\overline{g}$ et h supérieur à un certain h_0. Pour simuler efficacement un programme, le but est alors d'obtenir $L \geq gh$. Plus le réseau de communications est lent, plus le grain doit être élevé. Une phase de calculs locaux et de communications entre deux synchronisations s'appelle une *ronde*.

VALIANT montre comment ce modèle permet de simuler optimalement un programme EREW PRAM pour $v \geq p \log p$ en distribuant la mémoire de manière aléatoire sur les processeurs et gérant les accès à la mémoire grâce à une fonction de hachage [85, 59]. Il montre aussi comment simuler optimalement un programme CRCW PRAM pour $v \geq p^{1+\epsilon}$ (et $L \geq \log p$) en utilisant des méthodes parallèles de tri d'entiers [85, 84, 68]. Ces résultats sont théoriques et en pratique, il est important de minimiser le nombre de rondes de simulation de PRAM car elles impliquent en règle générale des tris et apportent une constante conséquente. Pour cela, il faut faire attention à la manière dont sont distribuées les données.

Le modèle CGM peut être vu comme une simplification du modèle BSP. Il permet de se concentrer sur le placement des données sans toutefois entrer trop dans le détail des communications. Lors du traitement d'un problème de taille N, tous les processeurs gèrent une partie des données comparable en taille et sont donc supposés disposer d'une mémoire locale en $O(N/p)$.

Une première simplification qui vaut au modèle la qualification de « gros grain » consiste à supposer le rapport N/p très supérieur à 1 ; typiquement, on suppose $N/p \geq p$, ce qui permet à chaque processeur de stocker un peu d'information concernant les autres processeurs (ne serait-ce qu'une table de routage vers chacun d'eux par exemple). Cette hypothèse est tout à fait réaliste en ce qui concerne les machines parallèles existantes : comme elles ont généralement moins de 1000 processeurs, un problème ne vérifiant pas cette hypothèse aurait une taille de quelques méga-octets, ce qui tient sur une machine séquentielle et ne justifie donc pas forcément l'utilisation d'une machine parallèle.

Une deuxième simplification consiste à supposer $h = N/p$. Elle consiste à supposer égales les complexités des différentes routines de communication que sont la diffusion («*broad-cast* » en anglais) de un vers tous, la diffusion de tous vers un, la diffusion personnalisée de tous vers tous, les sommes préfixées, et le tri. En effet, toutes ces opérations de communication peuvent être réalisées par un nombre constant de tris [26]. D'autre part, le tri par colonnes de LEIGHTON [54] permet de

trier $N \geq p^3$ nombres en cinq tris locaux entre lesquels des permutations simples des données sont effectuées. Le temps pris par une routine de communications sera noté $T_{com}(N,p)$. Remarquons que pour distribuer les données de manière efficace, il faut souvent faire au moins un tri. Dans le modèle CGM, il faut viser un nombre constant de rondes (et donc de tris) ou au pire $O(\log p)$ mais pas un nombre qui dépend de N de sorte que l'algorithme reste efficace pour une large variété de rapports N/p. Etant donné la puissance des ordinateurs séquentiels, on peut supposer N de l'ordre de 100 méga-octets (un problème polynômial sur une entrée de 10 méga-octets peut être résolu sur une machine séquentielle ; en revanche, un problème exponentiel est nécessairement petit si l'on peut le résoudre et l'hypothèse doit porter explicitement sur la mémoire locale disponible sur chaque processeur, on s'intéresse plutôt à des problèmes polynômiaux voire linéaires dans cette thèse). $\log N$ est donc supérieur à 27, et il n'est pas envisageable de trier $\log N$ fois un gros problème puisque cela fait un facteur supérieur à $27^2 \geq 700$[1].

Le modèle CGM relie donc de manière beaucoup moins subtile que le modèle BSP le rapport entre les calculs locaux et les communications. Cependant, il permet d'aborder de manière pertinente le placement des données sur une machine distribuée et fournit des algorithmes efficaces [26, 20, 31] pour une large variété de rapports N/p (cette propriété s'appelle «*scalability* » en anglais). Le modèle BSP permet en général d'affiner l'hypothèse $N/p \geq p$ en $N/P \geq p^\epsilon$ pour $\epsilon > 0$. En pratique, on a bien $N \geq p^2$ (en revanche, l'hypothèse $N \geq p^3$ serait plus discutable) et la simplification effectuée dans le modèle CGM est bien justifiée dans l'optique de concevoir des algorithmes d'implantation efficace.

Voici comment s'implanterait par exemple le calcul de sommes préfixées en CGM.

Calculs préfixés

Ici les sommes totales sur chaque processeur sont envoyées à tous les processeurs et chacun calcule la somme préfixée globale. Cela permet de ne faire qu'une seule ronde de communications. Selon la machine il peut être plus rapide de faire deux rondes : une première où chaque processeur envoie sa somme totale à P_0 et une deuxième où P_0 renvoie à chacun la somme préfixée des sommes totales des processeurs le précédant. Certaines machines comme la CM5 possèdent déjà une routine de calcul préfixé déjà implantée par le constructeur.

On comprend bien sur cet exemple du calcul préfixé comment l'hypothèse $n/p \geq p$ permet de faire le calcul en une seule ronde au lieu de $\log p$. L'algorithme 2.1 calcule les sommes préfixées d'une liste en temps $2n/p+p+T_{com}(n,p) = O(n/p + T_{com}(n,p))$.

1. Pour un très gros problème, $\log N$ est au plus de 50 (cela représente 1000 giga-octets sur chacun des 1000 processeurs d'une hypothétique machine), $\log^2 N$ est donc de l'ordre de 1000.

Algorithme 2.1 [26] Calcul préfixé

Données : Une liste L de n éléments distribuée sur les processeurs $P_0, \ldots, P_{p-1}$ telle que $n/p \geq p$.

Résultat : Une liste S des sommes préfixées $S(i) = L(1) \oplus \cdots \oplus L(i)$ où $\oplus$ est n'importe quelle opération associative.

Début

  Pour tout *processeur* P_α **effectuer**

    Calculer localement les sommes préfixées.

    Envoyer à tous les autres processeurs la somme totale T_α des éléments de P_α.

    Ajouter $T_0 \oplus \cdots \oplus T_{\alpha-1}$ aux sommes calculées précédemment.

Fin

Le tri

Le tri est l'outil de base par excellence du parallélisme. En effet, les données étant distribuées, il faut souvent les ranger de manière adéquate sur les différents processeurs. Le tri est un outil très puissant dans ce domaine car il permet d'effectuer des déplacements complexes de données sans se soucier de la topologie du réseau ou de la position initiale des éléments. Pour l'algorithmicien, c'est un outil très utile qui permet de faire abstraction du réseau de la machine cible, il suffit que le tri soit rapide. Pour le programmeur, le tri facilite l'implantation grâce à l'appel de cette routine de haut niveau.

Le tri est une routine de base de la plupart des machines parallèles. Toutefois il est possible d'implanter le tri avec les routines usuelles de communication. GOODRICH [38] a montré comment implanter le tri de n éléments sur $p < n^c$ processeurs ($c < 1$) avec $O(\log n / \log(h+1))$ rondes et avec un temps de calcul local de $O(n \log n/p)$ dans le modèle BSP. Dans le modèle CGM, cela se traduit par un nombre constant de rondes et un temps de calculs locaux optimal.

Remarquons que le tri du géomètre (ou «*bucket sort* » en anglais) qui permet de trier n éléments entre 0 et n^k en temps $O(nk)$ perd sa complexité linéaire en parallèle. Il y a une bonne raison à cela, ce tri utilise fortement l'accès direct à la mémoire offert par la RAM : il y a un $\log n$ caché dans le temps de tout accès à une case mémoire car ce tri fait l'hypothèse que $\log n$ est inférieur à la taille l des mots du processeurs (typiquement $l = 32$ ou 64 bits). Dans le cas du distribué, les n éléments à trier ne tiennent pas à priori dans la mémoire d'un seul processeur, et l'hypothèse $n \leq 2^l$ ne tient plus. On peut cependant supposer $n/p \leq 2^l$ (l'écriture binaire d'un nombre entre 0 et n occupant $\log p$ cases mémoires), ce qui donne un tri parallèle du géomètre avec un temps de calculs locaux en $O(\frac{nk}{p} \log p)$ si l'on remplace les tris locaux par des tris du géomètre.

Dans le cas où l'on trie les arêtes d'un graphe, on obtient un ainsi un tri

d'un temps de calculs locaux en $O(\frac{m}{p}\log p)$ avec un nombre constant de rondes. Cette remarque est intéressante en ce qui concerne certains problèmes linéaires en séquentiel car cela permet de ramener le temps des calculs locaux des tris de $O(\frac{n}{p}\log n)$ à $O(\frac{n}{p}\log p)$. On peut ainsi obtenir des algorithmes se situant à un facteur $\log p$ de l'optimal.

2.2 Algorithme de composantes connexes

Nous allons maintenant présenter en introduction au modèle comment calculer les composantes connexes d'un graphe dans le modèle CGM. La simulation de l'algorithme PRAM conduit à un algorithme en $O(\log n)$ rondes, ce qui est trop coûteux. Il est possible de ramener le nombre de rondes à $O(\log p)$.

Examinons tout d'abord dans un modèle distribué comment calculer localement les composantes connexes sur chaque processeur et essayer de combiner les résultats. Cela suppose que la mémoire de chaque processeur qui est de l'ordre de m/p peut contenir un numéro de composante pour chaque sommet, c'est-à-dire $m/p \geq n$. Cette hypothèse implique donc que le graphe a beaucoup d'arêtes.

Chaque processeur P_α, $0 \leq \alpha \leq p-1$ calcule séquentiellement une forêt couvrante des composantes connexes induites par les arcs qu'il a en mémoire. Les processeurs de numéro α impair envoient leur forêt au processeur $P_{\alpha-1}$. On recommence avec les processeurs $P_{2\alpha}$ qui peuvent maintenant calculer une forêt couvrante du graphe induit par les arcs stockés par $P_{2\alpha}$ et $P_{2\alpha+1}$, $1 \leq \alpha \leq P/2$, et ainsi de suite $\log p$ fois. Au bout du compte, P_0 a une forêt couvrante du graphe qu'il peut diffuser aux autres processeurs (par un «*broadcast* »).

Examinons maintenant le cas des graphes qui ont peu d'arêtes. Cette fois, c'est l'algorithme PRAM qui nous donne la solution, en nous permettant de nous ramener au cas précédent. Il suffit pour cela de simuler l'algorithme PRAM durant $\log p$ rondes pour qu'après contraction des arbres de pères, il y ait au plus n/p étoiles. Cette dernière opération de contraction est un sous-problème de «*list-ranking* » dont l'algorithme CGM est directement inspiré de l'algorithme PRAM optimal.

En combinant l'idée inspirée du séquentiel avec l'algorithme PRAM, on obtient l'algorithme CGM 2.2.

Remarquons que l'on obtient une forêt couvrante dans le cas où il y a peu d'arêtes en retrouvant pour chaque arête de la forêt couvrante de G' une arête de G dont elle est issue.

Comme l'article contenant ces résultats n'est pas encore publié[2] [29], la méthode pour calculer le list-ranking n'est pas détaillée ici. Les auteurs utilisent pour

2. A l'heure où j'écris ces lignes, ce modèle est tellement nouveau que les auteurs de ce résultat élémentaire n'ont pas encore écrit leur article. Cette note paraîtra sûrement amusante dans quelques années, que ce modèle connaisse un succès général ou un oubli total.

Algorithme 2.2 [29] Composantes connexes

Données : Les m arêtes d'un graphe $G = (V, \mathcal{E})$ tel que $\frac{n+m}{p} \geq p$.

Résultat : Les composantes connexes.

Début

Si $n \leq m/p$ **Alors**

Pour $t = 1$ **effectuer**

$\log p$

Pour tout $1 \leq \alpha \leq p$ *divisible par* 2^t **effectuer**

Calculer une forêt couvrante localement sur le processeur $P_{\alpha+2^{t-1}}$.
$P_{\alpha+2^{t-1}}$ envoie sa forêt à P_α.
P_α ajoute les arêtes de la forêt qu'il reçoit aux siennes.

{*Le processeur* P_0 *contient une forêt couvrante du graphe.*}
Le processeur diffuse à tous les processeurs le numéro de composante connexe $C(u)$ de chaque sommet u.

Sinon

Simuler $\log p$ pas de l'algorithme 1.1 où n/p cases du tableau *Père* sont distribuées à chaque processeur.
Effectuer un «*list-ranking* » sur le tableau *Père*.
Simuler l'instruction PRAM :
Pour tout *arête* $\widehat{ij}$ **effectuer**

Calculer l'arête $Père(i)\widehat{Père}(j)$ du graphe G'.
Effectuer cet algorithme sur le graphe G' qui a au plus $n/p \leq m/p$ sommets.

Chaque processeur calcule pour chacune des cases i du tableau *Père* :
$Père(i) \leftarrow C(Père(i))$.

Fin

cela une idée algorithmique assez jolie tirée de l'algorithme PRAM déterministe optimal.Cette technique est assez compliquée et je ne connais pas les détails de l'implantation en CGM. En revanche, on peut facilement imaginer un algorithme probabiliste pour résoudre ce problème. L'idée qui consiste à faire confiance au hasard pour sélectionner n/p éléments à peu prés équidistribués dans la liste est illustrée par l'algorithme 2.3.

Toute la difficulté réside dans l'analyse de l'algorithme qui est dans ce cas un grand classique d'algorithmique probabiliste. La probabilité pour que l'algorithme ne réussisse pas du premier coup est égale à la probabilité qu'il y ait une boîte contenant plus de $4p$ balles quand on lance n balles dans n/p boîtes. On peut montrer que cette probabilité est inférieure à $4n/p \left(\frac{e}{p}\right)^{4p}$ en s'inspirant du calcul page 44 de [63]. Pour $n = 10^{10}$ (plus de 10 giga-octets par processeur) et $p \geq 20$, cette probabilité est inférieure à $0,01$ L'algorithme s'exécute donc en $O(n/p +$

Algorithme 2.3 List-ranking probabiliste

Données : Les n éléments d'une liste L. Un tableau *Suivant* donne le successeur de chaque élément de L. On suppose $n/p \geq p$.

Résultat : Le rang $R(i)$ de chaque élément i de L.

Début

Chaque processeur sélectionne aléatoirement n/p^2 des éléments qu'il contient.

{*Si les données ne sont pas distribuées aléatoirement sur les processeurs, il est possible d'y remédier en une ronde : chaque processeur divise de manière aléatoire sa part d'éléments en p portions qu'il envoie à tous les processeurs selon une permutation aléatoire.*}

Initialiser R à 1.

Poser $R(d) \leftarrow 0$ et $Suivant(d) \leftarrow d$ pour le dernier élément d de L et pour tous éléments i sélectionnés, stocker la vieille valeur de $Suivant(i)$ dans $Suivant'(i)$.

Simuler $2 + \log p$ *opérations de «pointer-jumping »*

- **Si** $Suivant(i) \leftarrow Suivant(Suivant(i))$ **Alors**
  - $R(i) \leftarrow R(i) + R(Suivant(i))$
  - $Suivant(i) \leftarrow Suivant(Suivant(i))$

Si $Suivant'(Suivant(i))$ *est un élément sélectionné pour tout* i **Alors** continuer.

Sinon l'exécution a échoué, recommencer l'algorithme depuis le début.

Pour tout *élément* i *sélectionné* **effectuer**

- $Suivant(i) \leftarrow Suivant'(Suivant(i))$
- $R(i) \leftarrow R(Suivant(i)) + 1$

Chaque processeur envoie ses n/p^2 éléments sélectionnés et les valeurs associées au processeur P_0.

P_0 fait le list-ranking de la liste des éléments sélectionnés en séquentiel et renvoie à chaque processeur les rangs des sommets sélectionnés que celui-ci lui avait envoyé.

Simuler l'instruction :

Pour tout *élément* i **effectuer** $R(i) \leftarrow R(i) + R(Suivant(i))$

Fin

$\log p\, T_{com}(n,p))$ avec forte probabilité.

On attend généralement des algorithmes probabilistes séquentiels une probabilité de durée d'exécution supérieure à celle annoncée bornée par $1/n^c$. Il n'est pas aberrant de penser qu'un bon algorithme probabiliste parallèle présente une probabilité qui tend rapidement vers 0 quand le nombre de processeurs augmente.

2.3 Reconnaissance des ordres N-free

Nous allons maintenant aborder un algorithme CGM de reconnaissance des ordres N-free basé sur l'algorithme 1.11. Nous utiliserons les termes du paragraphe 1.3.

Les deux algorithmes PRAM 1.11 et 1.12 sont assez faciles à implanter en CGM. En effet, l'algorithme EREW 1.12 utilise un nombre constant de tris et l'algorithme CRCW 1.12 est en temps constant. En fait, la simple simulation de ce dernier donne un algorithme CGM ayant un nombre constant de rondes, ce qui montre l'utilité des algorithmes PRAM en temps constant.

L'algorithme 2.4 est une version CGM de l'algorithme EREW 1.12.

Algorithme 2.4 Reconnaissance des ordres N-free

Données : Les m arêtes d'un graphe orienté sans circuit $D = (V, \mathcal{A})$ réduit transitivement tel que $\frac{n+m}{p} \geq p$.

Résultat : Un diagramme d'arcs si la fermeture transitive de D est un ordre N-free.

Etape 1

Trier les arcs anti-lexicographiquement.
Associer à chaque arc par un calcul préfixé un numéro de composante bipartie : l'origine du premier arc ayant même destination.
Stocker pour chaque bloc d'arcs de même destination et donc de même numéro de composante bipartie b l'information $Arc(v) = b, \infty$.

Etape 2

Trier les arcs lexicographiquement.
Vérifier par un calcul préfixée que les arcs ayant même origine ont même numéro de composante et calculer en même temps le nombre d'arcs dans chaque bloc de même origine.
Stocker pour chaque bloc d'arcs de même origine u et de même numéro de composante bipartie b les informations $Arc(u) = 0, b$ et $b, Degré^+(u)$.
Trier les couples $b, Degré^+(u)$ par b croissant et vérifier par un calcul préfixé que tous ceux qui ont même première composante ont même deuxième composante.

Construction d'un diagramme d'arcs.

Chaque processeur envoie les enregistrements $Arc(u)$ qu'il possède au processeur $P_{\lfloor u/p \rfloor}$.
Chaque processeur P_α calcule l'arc du diagramme d'arcs associé à chaque sommet $\alpha n/p + 1 \leq u \leq (\alpha + 1)n/p$: s'il reçoit zéro, un ou deux enregistrements le concernant, l'arc sera respectivement $0, \infty$, ou d'une des deux formes b, ∞ ou bien $0, b$, ou de la forme b, b'.

Les « *broadcast* » sont en fait des calculs préfixés et peuvent être faits par une

diffusion de tous vers P_0 qui calcule ce dont chaque processeur a besoin et leur envoie. Il faudrait tester ce qui est le plus rapide en pratique. La méthode donnée dans l'algorithme 2.4 prend une ronde avec p^2 messages, la méthode évoquée ici prend deux rondes avec $2p$ messages.

Théorème 32 *L'algorithme 2.4 détermine si la réduction d'un graphe réduit transitivement est un ordre N-free, et en calcule un diagramme d'arcs dans l'affirmative, en temps* $O(\frac{n+m}{p}\log p + T_{com}(n+m,p))$.

2.4 Représentation compacte des ordres de dimension d

Il existe déjà une version CGM de la gestion d'une base de données d'éléments d'un espace de dimension d fixée [30], en vue de la réponse à des requêtes de type intervallaire : « Quels sont les éléments de base de données qui ont tel caractère entre telle et telle valeur, tel autre caractère dans tel autre intervalle ...? ».Cet article est basé sur la structure de «*range tree* ». Cette structure de données est très récursive et par conséquent difficile à comprendre, ce qui n'en facilite pas l'implantation.

Nous allons voir comment implanter la structure de données compacte introduite dans le paragraphe 1.2 dans le modèle CGM. Nous étudierons tout d'abord la simulation de l'algorithme PRAM avant d'aborder une version minimisant le nombre de rondes. Pour simplifier les notations, nous supposerons que n et p sont des puissances de 2 (si ce n'est pas le cas, il faut rajouter des parties entières là où l'on n'a plus des entiers).

Simulation de l'algorithme pram

L'implantation la plus simple consiste à simuler l'algorithme PRAM 1.8.

L'idée la plus simple s'avère la mieux adaptée : distribuer les tableaux W_f de manière triée sur les processeurs : les n/p premiers éléments sur P_0, les n/p suivants sur P_1, etc... Chaque élément $W_f(x) = v$ est associé à un intervalle $I_f(x)$ de W_f et aussi à une copie de $(\pi_1(v), \ldots, \pi_d(v))$.

Remarquons que la simulation du tri « à la quicksort » effectuera $\log p$ rondes et non pas $\log n$. En effet, la taille des blocs vaut n/p après $\log p$ phases, aussi les éléments de $W_{f,\log p}, \ldots, W_{f,\log n}$ sont distribués de la même manière sur les processeurs, et seul l'ordre des éléments à l'intérieur de chaque mémoire locale change. Comme il n'est pas nécessaire de faire des copies d'un même élément à l'intérieur de chaque processeur, la taille de la représentation sera donc de l'ordre de $O(n\log^{d-1} n + dn\log^{d-1} p)$. Pour $d \geq 3$ et pour des valeurs réalistes de n et p c'est-à-dire $\log n \geq 23$ et $\log p \leq 15$, on a toujours $d\log^{d-1} p \leq \log^{d-1} n$ et l'espace

mémoire utilisé est optimal à un facteur 2 près[3].

Voir l'algorithme 2.5.

Algorithme 2.5 Calcul d'une représentation compacte

Données : n éléments $v(0) = (\pi_1(0), \ldots, \pi_d(0)), \ \ldots, \ v(n-1) = (\pi_1(n-1), \ldots, \pi_d(n-1))$.

Résultat : Un ensemble $\mathcal{W}$ de $\log^{d-1} n$ tableaux et les intervalles de successeurs $I_f(x) = [g_f(x), d_f(x)]$ associés à chaque sommet $v(W_f(x))$.

Début

Trier les éléments suivant la première coordonnée pour obtenir le tableau W_1.
Pour tout $0 \leq x < n$ **effectuer** $I_1(x) \leftarrow [x+1, n-1]$
Poser $\mathcal{W} = \{W_1\}$.
Pour $\delta = 2$ *à* d **effectuer**
  Pour tout $W_f \in \mathcal{W}$ **effectuer**
    Simuler $\log p$ phases du calcul d'une représentation compacte de l'intersection de l'ordre de contiguïté représenté par W_f et I_f avec l'ordre total induit par π_δ en triant W_f « à la quicksort » selon la δ^{e} coordonnée selon l'algorithme 1.8.
    On obtient ainsi les $\log p$ premiers tableaux de la représentation compacte $W_{f,0}, \ldots, W_{f,\log p}$ et $I_{f,0}, \ldots, I_{f,\log p}$.
    Continuer séquentiellement sur chaque processeur le calcul de $W_{f,\log p+1}, \ldots, W_{f,\log n}$ et les intervalles associés $I_{f,\log p+1}, \ldots, I_{f,\log n}$.
  Poser $\mathcal{W} \leftarrow \bigcup_{W_f \in \mathcal{W}} \{W_{f,0}, \ldots, W_{f,\log n}\}$.

Fin

Théorème 33 *L'algorithme 2.5 calcule une représentation intervallaire compacte d'un ordre de dimension d donné par d permutations en temps* $O(\frac{n \log^{d-1} n}{p} + (d \log p)\, T_{com}(dn \log^{d-2} n, p))$.

Remarquons que cet algorithme est tout à fait acceptable de par sa simplicité et son nombre de rondes relativement faible. En revanche, dans l'application en base de données, la simulation de l'algorithme 1.9 risque de donner de mauvais résultats dans le cas où les requêtes ne se distribuent pas équitablement lors de l'insertion dans les tableaux : un seul processeur risque de récupérer toutes les requêtes dans son bloc de taille n/p après $\log p$ phases de simulation.

3. Ces bornes sont vraiment larges car 2^{23} représente environ 10 méga-octets, ce qui tient sur un ordinateur personnel et $2^{16} \simeq 64000$, ce qui est le nombre de processeurs du plus gros ordinateur parallèle jamais construit, la «*connexion machine* ».

Calcul des intervalles de nouveaux sommets

Nous allons voir dans ce paragraphe comment calculer les intervalles de $n' = O(n)$ nouveaux sommets avec $d \log p$ rondes malgré tout. Ce problème est la traduction sur l'ordre de dimension d de la réponse à des requêtes dans le contexte des bases de données. Notons $\mathcal{W}_{\log p}$ l'ensemble des tableaux W_f de la représentation compacte qui ont des blocs de taille n/p. La simulation de l'algorithme PRAM 1.9 permet de calculer les intervalles des nouveaux sommets des tableaux de taille de bloc supérieure à n/p ; il reste ensuite à poursuivre les calculs localement sur les processeurs à l'intérieur des blocs de taille inférieure à n/p. Le problème consiste à équilibrer ces calculs sur les processeurs. Voir l'algorithme 2.6.

Pour ce qui est des envois de parties de la représentation compacte, au pire chaque processeur envoie toute sa partie locale de la représentation et en reçoit autant. Chaque processeur reçoit au plus $2n/p$ requêtes par tableau $W_g \in \mathcal{W}_{\log p}$. Il peut donc subsister un déséquilibre entre les processeurs vis à vis du nombre de requêtes qu'ils ont à traiter lorsque n' est très inférieur à n.

Théorème 34 *L'algorithme 2.6 permet de trouver les intervalles de n' requêtes en temps $O(\frac{(n+n')\log^{d-1} n}{p} + (d \log p)\, T_{com}(dn \log^{d-2} n, p))$.*

Dans [30], les requêtes restent équilibrées sur les processeurs mais c'est la base de données qui peut être complètement dupliquée. La solution adoptée ici permet de garder des communications faibles dans le cas où n' est petit par rapport à n et demande le même temps de calcul dans le pire cas. Les deux méthodes sont efficaces lorsque n' est de l'ordre de n (dans ce cas, le plus rapide est peut-être de reconstruire toute la base de données pour les $n + n'$ sommets) ou lorsque les requêtes sont équidistribuées (si elles sont supposées aléatoires par exemple).

Calcul du nombre d'arcs de l'ordre

Pour calculer le nombre d'arcs de l'ordre, il suffit de rassembler tous les intervalles correspondant à chaque sommet. Voir l'algorithme 2.7.

Théorème 35 *L'algorithme 2.7 calcule en temps $O(\frac{n \log^{d-1}}{p} + T_{com}(n \log^{d-1} n, p))$ le nombre d'arcs d'un ordre de dimension d à partir d'une représentation compacte.*

Calcul des arcs de l'ordre

Une fois que l'on a calculé les intervalles de successeurs de chaque sommet, on peut calculer à l'avance comment placer les listes d'adjacence des sommets pour qu'elles soient équidistribuées sur les processeurs et en particulier à quel processeur envoyer chaque sommet d'un intervalle de successeurs. Le calcul du nombre de prédécesseurs de chaque sommet permet de calculer le nombre de

Algorithme 2.6 Calcul des intervalles de nouveaux sommets

Données : Les tableaux W_f et I_f d'une structure de données compacte représentant un ordre de dimension d. $n' = O(n)$ nouveaux sommets $u(0) = (\pi'_1(0), \ldots, \pi'_d(0)), \ldots, u(n-1) = (\pi'_1(n'-1), \ldots, \pi'_d(n'-1))$ où les $\pi'_\delta(i)$ sont des nombres entre 0 et $n-1$.

Résultat : Les intervalles de successeurs $I'_f(i)$ associés à chaque nouveau sommet $u(i)$.

Début

Pour tout *processeur* P_α **effectuer**

Simuler $\log p$ phases de l'algorithme 1.9 pour chaque sommet $u(i)$, $\alpha n'/p \leq i < (\alpha+1)n'/p$.

Pour chaque sommet $u(i)$, retenir le rang x où il s'insère dans chaque tableau $W_g \in \mathcal{W}_{\log p}$ ainsi que l'intervalle I associé, et créer une requête élémentaire $r(i) = (x, g, I)$.

Trier les requêtes lexicographiquement.

Soit R_g^β l'ensemble des requêtes élémentaires (x, g, I) telles que $\beta n/p \leq x < (\beta+1)n/p$.

Pour tout *processeur* P_α **effectuer**

Calculer localement le nombre N_g^β de requêtes dans R_g^β sur P_α.

Si $N_g^\beta = n/p$ **Alors**

Recevoir du processeur P_β une copie de la partie de la représentation compacte obtenue à partir du β^{e} bloc du tableau W_g.

{*Le processeur P_β enverra un seul message destiné à l'intervalle des processeurs lui demandant la partie de la représentation compacte en question.*}

Sinon

Envoyer les requêtes $r \in R_g^\beta$ au processeur P_β .

{*Le processeur P_β recevra au plus $2n/p$ requêtes concernant le tableau W_g.*}

Faire localement les calculs concernant chaque requête reçue et chaque requête gardée.

Envoyer les résultats concernant chaque requête $r(i)$ au processeur $\lfloor i/p \rfloor$.

Fin

copies à faire pour chaque futur successeur. Ces remarques permettent de calculer les arcs de l'ordre en temps $O(\frac{m+n\log^{d-1}}{p} + T_{com}(m + n\log^{d-1}, p))$.

Algorithme 2.7 Calcul du nombre d'arcs

Données : Les tableaux W_f et I_f d'une structure de données compacte représentant un ordre de dimension d.
Résultat : Le degré de chaque sommet et le nombre total d'arcs.
Début
  Pour tout *processeur* P_α **effectuer**
    Envoyer au processeur $P_{\lfloor v/p \rfloor}$ une copie de chaque intervalle $I_f(x)$ stocké en mémoire locale en posant $v = W_f(x)$.
    P_α reçoit tous les intervalles des sommets x tels que $\alpha n/p \leq x < (\alpha+1)n/p$.
    Calculer pour chacun de ses sommets la somme des longueurs des intervalles associés pour trouver son degré.
  Faire une somme préfixée sur les degrés pour trouver le nombre total d'arcs.
Fin

Implantation CGM

CGM privilégie un nombre faible de rondes. En pratique cela signifie qu'il vaut mieux envoyer peu de longs messages que beaucoup de courts. Dans ce problème, on peut très bien appliquer cette philosophie pour effectuer une seule ronde au lieu des $d \log p$ de la simulation. Etant donné la nouveauté du modèle, il faudrait vérifier qu'en pratique c'est effectivement plus rapide de faire quelques rondes de communications compliquées plutôt que $\log p$ rondes de communications simples.

Dans le modèle CGM le tri demande un nombre constant de rondes, c'est pourquoi les $\log p$ rondes imposées par le tri « à la quicksort » sont superflues. Etant donné W_f, on peut obtenir les tableaux $W_{f,0}, \ldots, W_{f,\log p}$ en un seul tri. On peut même faire cela pour tous les tableaux W_f en un seul tri un peu plus complexe. Pour cela, chaque processeur duplique lui-même chaque élément $v = W_f(x)$ de son morceau de W_f en $\log p + 1$ copies $v_{f,\phi} = g_{\phi-1}(\pi_\delta(v)), f, \phi, x,\ 0 \leq \phi \leq \log p$. Chaque élément contient aussi une copie de l'information $v = (\pi_1(v), \ldots, \pi_d(v))$. $g_{\phi-1}(\pi_\delta(v))$ donne le bloc qui contient v à la ϕ^{e} phase du tri « à la quicksort » selon la δ^{e} coordonnée. Rappelons la définition $g_\phi(y) = b_1 \ldots b_\phi 10 \ldots 0$ quand la représentation binaire de y est $y = b_1 \cdots b_q$ (on pose $g_{-1}(y) = 0$). Il suffit ensuite de trier lexicographiquement les quadruplets $v_{f,\phi}$ pour obtenir les tableaux voulus de la représentation compacte. Remarquons que les sommes préfixées $S_{f,\phi}$ ne sont pas calculées ici.

Rien n'empêche non plus d'appliquer le même principe pour les tableaux des dimensions suivantes et de construire la structure de données compacte avec un seul tri. Cela donne l'algorithme suivant :

Cet algorithme tient en une seule ronde. Mais dans le cas de l'application aux bases de données ou de l'application en géométrie à des points dans un espace de

Algorithme 2.8 Calcul d'une représentation compacte

Données : n éléments $v_0 = (\pi_1(v_0), \ldots, \pi_d(v_0)), \ \ldots, \ (\pi_1(v_{n-1}), \ldots, \pi_d(v_{n-1}))$.

Résultat : Les tableaux W_f de la représentation intervallaire compacte de l'ordre de dimension d associé.

Début

Pour tout *processeur* P_α **effectuer**

Pour tout *élément* v *de* P_α **effectuer**

Créer $\log^{d-1} p$ éléments de la représentation compacte :
$v_{\phi_2,\ldots,\phi_d} = g_{\phi_d - 1}(\pi_d(v)), \ldots, g_{\phi_2 - 1}(\pi_2(v)), \phi_d, \ldots, \phi_2, \pi_1(v)$
pour tout $1 \leq \phi_2, \ldots, \phi_d \leq \log p$. {*On peut se restreindre aux* $\phi_2, \ldots, \phi_d$ *tels que un* ϕ_i *au plus vaut* $\log p$}

Trier lexicographiquement les $n \log^{d-1} p$ éléments créés. {*Une comparaison de deux éléments prend un temps* $O(d)$}

Chaque processeur reçoit un morceau de chaque tableau $W_{\phi_2,\ldots,\phi_d}$ constitué des éléments de mêmes $2(d-1)$ premières composantes.

Continuer localement sur chaque processeur le calcul des tableaux restants à partir des $W_{\phi_2,\ldots,\phi_d}$ tels que l'un des ϕ_δ vaut $\log p$ (la taille des blocs y est donc inférieure à n/p).

Fin

dimension d, les π_δ ne sont plus des permutations mais de simples tableaux de nombres (entiers ou réels). Il faut alors modifier le calcul des $g_\phi(\pi_\delta(v))$. Un premier tri des d tableaux $\pi_1, \ldots, \pi_d$ permet d'en obtenir les pivots, seuls les pivots des $\log p$ premières phases nous intéressent, cela fait $1 + 2 + 4 + \cdots + 2^{\log p - 1} = p - 1$ pivots par coordonnée. L'algorithme fait donc l'hypothèse $n/p \geq p$. La procédure BlocsCourants permet ensuite de calculer $g_0(\pi_\delta(v)), \ldots, g_{\log p - 1}(\pi_\delta(v))$.

Pour calculer la représentation compacte, il reste encore à calculer les intervalles associés à chacun des éléments des W_f. On peut utiliser pour cela une technique similaire à celle employée dans le calcul des tableaux W_f. Cette technique permet tout aussi bien de calculer les intervalles associés à des nouveaux éléments (c'est-à-dire à des requêtes dans le contexte des bases de données). Nous allons donc traiter ce problème plus général.

Traitement des requêtes dans une base de données distribuée

La technique consiste à calculer localement des éléments fictifs représentant les bornes des différents intervalles d'un sommet donné. On peut ensuite trouver grâce à un tri puis une somme préfixée le rang des positions où ces bornes s'insèrent dans les tableaux W_f. Voir l'algorithme 2.10.

Cet algorithme n'est à nouveau efficace que lorsque le nombre de requête est de l'ordre de la taille de la base de données. Le déséquilibre qu'il peut y avoir

Algorithme 2.9 Calcul des $g_\phi(\pi_\delta(v))$

Données : d tableaux de n nombres $\pi_1, \ldots, \pi_d$.
Résultat : Les pivots des $\log p$ premières phases.
Pré-calcul des pivots
Trier les couples $(\delta, \pi_\delta(v))$ lexicographiquement.
Calculer par une somme préfixée le rang de chaque élément dans chaque tableau π_δ.
Pour chaque tableau π_δ, diffuser à tous les processeurs les éléments ψ_δ^r de rang $r = n/2, n/4, 3n/4, \ldots$ (r est de la forme $2^{\log n - \log p} \times$ un nombre de $\log p$ bits).
Procédure *BlocsCourants*$(\pi_\delta(v))$
Début
Poser $g_{-1}(\pi_\delta(v)) \leftarrow 0$.
Pour $\phi \leftarrow 0$ *à* $\log p - 1$ **effectuer**
$r \leftarrow g_{\phi-1}(\pi_\delta(v))$ où le $\phi + 1^{\text{e}}$ bit est mis à 1.
Si $\pi_\delta(v) < \psi_\delta^r$ **Alors**
$g_\phi(\pi_\delta(v)) \leftarrow g_{\phi-1}(\pi_\delta(v))$
Sinon $g_\phi(\pi_\delta(v)) \leftarrow r$.
Fin

sur la destination des différentes requête paraît difficile à gérer quand il y a peu de requête. En pratique, ce cas n'est probablement pas gênant si l'on utilise un algorithme du type de l'algorithme 2.6, s'il y a peu de requête, un seul processeur pourra les traiter assez rapidement.

Nous allons maintenant aborder la partie traitant des graphes de comparabilité.

2.5 Reconnaissance des graphes de comparabilité

Une fois de plus, l'algorithme PRAM (voir le paragraphe 1.4) s'implante assez facilement mais sa complexité en fait un algorithme peu efficace. Toutefois il pourrait permettre de traiter des graphes trop gros pour rentrer dans la mémoire d'un ordinateur séquentiel. Mais là aussi, sa complexité (en mémoire) le dessert car la relation de forçage direct prend une place mémoire en $O(\delta m)$, ce qui risque d'être trop gros même pour une machine parallèle (c'est encore acceptable pour les graphes de degré borné, ce qui est très restreint).

Pour obtenir un algorithme utilisable, il faut utiliser un espace mémoire total en $O(n+m)$. Pour identifier les classes d'implication efficacement, il faut comme en séquentiel calculer les composantes connexes à chaque pas du calcul de la

Algorithme 2.10 Evaluation de la taille des requêtes

Données : Les tableaux W_f de la représentation compacte associée à n d-uplets. $n' = O(n)$ d-uplets de requêtes.

Résultat : Les intervalles et la taille de chaque requête.

Début

Pour tout *processeur* P_α **effectuer**

Pour tout *requête* v *de* P_α **effectuer**

Créer $2\log^{d-1} p$ bornes d'intervalles de v :

$v^g_{\phi_2,\dots,\phi_d} = g'_{\phi_d-1}(\pi_d(v)), g_{\phi_{d-1}-1}(\pi_{d-1}(v)), \dots, g_{\phi_2-1}(\pi_2(v)), \phi_d, \dots, \phi_2, \pi_1(v)+1$

$v^d_{\phi_2,\dots,\phi_d} = g'_{\phi_d-1}(\pi_d(v)), g_{\phi_{d-1}-1}(\pi_{d-1}(v)), \dots, g_{\phi_2-1}(\pi_2(v)), \phi_d, \dots, \phi_2, \infty$

pour tout $1 \leq \phi_2, \dots, \phi_d \leq p$ tel que le $\phi + 1^{\text{e}}$ bit de $g_{\phi_d-1}(\pi_d(v))$ vaut 0 et où $g'_{\phi_d-1}(\pi_d(v))$ a les mêmes bits que $g_{\phi_d-1}(\pi_d(v))$ sauf le $\phi + 1^{\text{e}}$ qui est mis à 1. {*On peut se restreindre aux* $\phi_2, \dots, \phi_d$ *tels que un* ϕ_i *au plus vaut* $\log p$}

Trier les tableaux $W_{\phi_2,\dots,\phi_d}$ et les bornes pour trouver les rangs des bornes des intervalles.

Continuer les calculs dans les représentations locales de tailles de bloc inférieures à n/p.

Fin

relation de forçage.

Le calcul de la relation de forçage ressemble beaucoup au produit de matrices. En effet, pour chaque arête $\widehat{uv}$, il faut considérer les arêtes $\widehat{vw}$ telles que $\widehat{uw}$ n'est pas une arête du graphe. Il n'est cependant pas étonnant de trouver une procédure qui ressemble à la fermeture transitive dans un algorithme de calcul d'une orientation fermée transitivement.

Nous allons présenter un algorithme qui utilise un espace mémoire en $O(\delta n)$, ce qui dans le pire cas revient à avoir une représentation matricielle du graphe. Chaque processeur traite les listes d'adjacence de n/p sommets. On suppose donc $\delta n/p \geq \delta$, c'est-à-dire $n \geq p$. L'algorithme est en deux temps : par une procédure du type produit de matrice, chaque processeur construit pour chacun de ses sommets une forêt couvrante des relations de forçage des arcs entrants du sommet et une autre sur les arcs sortants. Le deuxième temps consiste à faire l'union de toutes les forêts couvrantes. Les classes d'implication sont les composantes connexes de cette union de forêts qui a une taille inférieure à $2m$.

Théorème 36 *L'algorithme 2.11 détermine les classes d'implications d'un graphe (tel que* $n \geq p$*) en temps* $O(\frac{\delta n^2}{p} + (p + \log p)\, T_{com}(n\delta, p))$.

Concernant l'espace mémoire, l'hyptothèse la plus forte de l'algorithme consiste

Algorithme 2.11 Calcul des classes d'implication

Données : Les listes d'adjacence de chaque sommet d'un graphe $G = (V, \mathcal{E})$.
Résultat : Le numéro de classe d'implication de chaque arc.
Etape 1 *Relations de forçages « locales »*
- Chaque processeur P_α possède un bloc L_α de listes d'adjacence et en fait une copie K_α.
- **Effectuer** p *rondes*
  - **Pour tout** *processeur* P_α **effectuer**
    - **Pour toute** *liste d'adjacence* A *de* L_α *d'un sommet* u **effectuer**
      - **Pour tout** *voisin* v *de* u *dont la liste d'adjacence* B *est dans* K_α **effectuer**
        - Calculer $A - B$.
        - **Pour tout** $w \in A - B$ **effectuer**
          - **Si** uw *n'est pas dans le même arbre que* uv **Alors**
            - Rajouter la relation de forçage direct $uw \sim uv$ à la forêt des arcs sortants de u.
            - Rajouter de même $wu \sim vu$ la forêt des arcs entrants.
      - Recalculer les composantes connexes des deux forêts.
    - Envoyer K_α au processeur suivant : $K_\alpha \leftarrow K_{\alpha-1 \bmod p}$.
- Calculer les composantes connexes de l'union des forêts.

à supposer que chaque liste d'adjacence tient dans la mémoire d'un processeur. L'autre hypothèse sur la taille de la mémoire ne servait qu'à expliquer l'algorithme en termes de produit de matrice. En effet, on peut découper l'ensemble des arcs du graphe en blocs de listes d'adjacence de taille $2m/p$ au plus.

Si l'on ne fait plus l'hypothèse que la liste d'adjacence d'un sommet tient toujours dans la mémoire d'un processeur, la même technique peut toujours s'appliquer mais il faut « recoller les morceaux » d'une ronde à l'autre. Il faut commencer par trier les arcs lexicographiquement pour que la liste d'adjacence d'un tel sommet soit répartie sur des sommets consécutifs. Il y a deux problèmes à résoudre : celui d'un processeur qui gère un tel sommet et celui d'un processeur qui travaille sur un K_α qui n'est qu'une partie de liste d'adjacence. Dans ce dernier cas, il suffit de continuer le calcul $A - B$ à la ronde suivante quand la suite de la liste d'adjacence arrive avec $K_{\alpha-1}$ (les listes étant triées, cela ne pose pas de problème en faisant ce calcul par fusion des deux listes).

Le cas où la liste d'adjacence d'un sommet u stockée sur le processeur P_α se poursuit sur le processeur suivant est plus délicat. Quand le processeur P_α a identifié un voisin de u, il faut qu'il passe cette information au processeur

suivant avec K_α pour que les processeurs suivants ne manquent pas de relation de forçage. P_α envoie donc à $P_{\alpha+1}$ une copie de l'arête $\widehat{uv}$ avec K_α pour chaque voisin[4]. (Remarquons que P_α possède au plus un sommet dont la liste d'adjacence se prolonge sur $P_{\alpha+1}$).

Il faut continuer à maintenir les composantes connexes de la forêt sur les arcs sortants de u sur tous les processeurs qui la contiennent, pour que le nombre de relations de forçages stockées sur les processeurs de plus grand numéro n'explose pas. La seule façon que nous connaissions actuellement pour réaliser ce genre d'opérations consiste à faire un calcul de composantes connexes à chaque ronde. Certains arcs sortants de u sont dupliqués et ont été passés avec K_α. Les copies servent à calculer les forêts locales de relations de forçage. On pourrait croire que l'on manque des relations de forçages du type $uv \sim uw$ lorsque $v > w$ sont dans la liste d'adjacence de u, et v se trouve sur un processeur de plus grand numéro que celui sur lequel se trouve w. Si cette relation de forçage passe inaperçue quand c'est la liste d'adjacence de v qui est inspectée à travers les K_α, elle est en revanche considérée quand c'est la liste d'adjacence de w qui est inspectée.

Il y a au plus une copie par arc. L'algorithme des composantes connexes s'exécute donc sur un graphe de $\mathit{Degré}(u)$ sommets (les arcs sortants de u), et au plus $2\,\mathit{Degré}(u)$ arêtes (celles des forêts locales de relations de forçage). Au total, il aboutit au bout d'un temps $O(\frac{m \log p}{p} + \log p\, T_{com}(m,p))$.

Les données risquent d'être distribuée de manière hétérogène : toutes les listes d'adjacence d'un processeur P_α peuvent être fusionnées avec toutes les listes de K_α, ce qui prend $\sum_{u,v} |L_u| + |M_v|$ opérations où L_u désigne la liste d'adjacence locale de u et M_v la liste d'adjacence de K_α d'un voisin $v \in L_u$. Ceci conduit à une borne en $\frac{m^2}{p^2}$ qui est atteinte lorsqu'un processeur n'a qu'une seule liste d'adjacence L_u et que K_α est composé de $\frac{m}{p}$ listes singletons de sommets qui sont tous dans L_u. La seule borne décroissante avec p que l'on peut obtenir est donc $O(\frac{m^2}{p})$. Ce dernier problème vient du fait que l'on a mal réparti les données. Pour le résoudre, il faut distribuer les listes d'adjacence de sorte que chaque processeur reçoive de l'ordre de n/p listes. Il faut pour cela traiter différemment les sommets de degré supérieur à m/p (dont la liste d'adjacence ne tient pas dans la mémoire locale d'un processeur) et les autres. De tels sommets sont en nombre au plus $p-1$. On place leur listes par un tri lexicographique par exemple, chaque processeur aura au plus deux morceaux de listes. Pour placer les autres listes d'adjacence, on trie les sommets restants par degré décroissant et on place ainsi : la liste du premier sommet sur P_0, la liste du deuxième sommet sur P_2,..., la liste du i^{e} sommet sur $P_{i-1 \bmod p}$. Avec cette stratégie, chaque processeur aura au plus n/p listes. C'est le processeur P_0 qui stocke le plus d'arcs. En revanche, si on lui enlève la première et la dernière liste que l'on lui a alloué, il lui reste moins d'arcs que sur chacun des autres processeurs, ce qui fait donc moins de m/p arcs. Au total, le processeur P_0 stocke donc au plus $3m/p$ arcs et les autres processeurs en

4. « voisin de u » signifie ici « dans le morceau de la liste d'adjacence de u stockée dans P_α ».

ont moins. En comptant aussi les sommets de degré supérieur à m/p, on obtient une distribution des données telle que chaque processeur a au plus $n/p+2$ listes d'adjacence et au plus $4m/p$ arcs.

Revenons à l'algorithme de calcul des classes d'implication lorsque les données sont distribuées ainsi. On peut toujours l'exécuter car sur chaque processeur, il y au plus une liste d'adjacence qui se poursuit sur le processeur suivant[5]. On peut maintenant borner le calcul local de chaque ronde :

$$\sum_{u,v} |L_u| + |M_v| \leq \sum_{u,v} |L_u| + \sum_{u,v} |L_v| \leq 2\frac{n}{p}\frac{m}{p}$$

. On obtient donc le résultat suivant.

Théorème 37 *L'algorithme 2.11 avec les adaptations ci-dessus calcule les classes d'implication d'un graphe G tel que $m/p \geq p$ en temps $O(\frac{m \log n}{p} + \frac{nm}{p} + m \log p + p \log p\, T_{com}(m,p))$.*

La borne en δm est devenue nm, ce qui permet de mettre en évidence les problèmes que posent l'irrégularité des structures de données optimales des graphes dans leur traitement parallèle. Autoriser l'algorithme PRAM 1.13 à utiliser un espace mémoire en $O(\delta m)$ masquait cette difficulté. Nous voyons ici que la recherche d'un temps polylogarithmique pour un algorithme PRAM n'est pas justifiée dans le cas où le travail de l'algorithme dépasse largement l'ordre de grandeur de la taille du problème. Il faut concevoir des algorithmes PRAM utilisant un nombre de processeurs de l'ordre de la taille du problème.

Les algorithmes d'orientation 1.14 et de décomposition modulaire 1.16 s'implantent facilement en CGM puisqu'ils utilisent un nombre constant de tris, de calculs préfixés et de calculs de composantes connexes.

2.6 Les graphes creux

L'exemple du traitement des graphes de comparabilité nous permet de mettre en évidence la difficulté de traiter les graphes creux, c'est-à-dire ayant peu d'arêtes. L'hypothèse $m \geq p\delta$ simplifie le calcul des classes d'implication. On a vu comment dans les composantes connexes l'hypothèse $m \geq pn$ simplifiait la tâche. Considérons à ce propos un problème très utile et très simple dans le domaine des graphes : calculer le degré de chaque sommet.

En séquentiel, on fait l'hypothèse que le graphe tient en mémoire et donc sans s'en rendre compte, on suppose que l'on peut garder en permanence en mémoire un compteur pour chaque sommet du graphe. Cette hypothèse pourrait très bien s'avérer fausse si le graphe est si gros qu'il nécessite l'utilisation de mémoire

5. Cette remarque tient aussi pour les K_α avec la remarque qu'il faut faire une ronde supplémentaire pour les processeurs P_α tels que K_α contenait initialement une fin de liste d'adjacence.

virtuelle sur le disque dur. Et dans ce cas, le meilleur algorithme risque fort de ressembler à un algorithme parallèle.

En effet, le parallélisme nous confronte directement à cette hypothèse. Considérons le cas où l'on peut encore stocker dans la mémoire de chaque processeur un compteur pour chaque sommet, c'est-à-dire le cas où $m/p \geq n$. Dans ce cas l'algorithme séquentiel peut encore servir : calculer localement les degrés sur chaque processeur et faire la somme ensuite. L'algorithme 2.12 en donne les détails.

Algorithme 2.12 Calcul des degrés

Données : m arcs d'un graphe, chaque processeur gère m/p arcs.

Résultat : Les degrés de tous les sommets, chaque processeur gère n/p sommets.

Début

  Pour tout *processeur* P_α **effectuer**

    Parcourir la liste locale d'arcs et calculer le degrés $d_\alpha(i)$ de chaque sommet rencontré.

    Pour tout i envoyer $d_\alpha(i)$ au processeur $P_{\lfloor i/p \rfloor}$.

    Pour tout $\alpha n/p \leq i < (\alpha+1)n/p$ **effectuer**

      Calculer $Degré(i) \leftarrow d_0(i) + \cdots + d_{p-1}(i)$.

Fin

L'algorithme 2.12 calcule les degrés des sommets d'un graphe en temps $O(n + m/p + T_{com}(pn, p))$. Un processeur peut très bien recevoir de l'ordre de n messages. Cet algorithme n'est optimal que lorsque $m/p \geq n$, c'est-à-dire lorsque le graphe est dense ou lorsqu'il y a peu de processeurs (si m est de l'ordre de n, cela peut vouloir dire un seul processeur).

D'un autre côté, le modèle PRAM s'intéresse au cas où la mémoire de chaque processeur est très restreinte, et il offre donc naturellement des solutions lorsque des hypothèses trop fortes sur la taille du graphe ou sur le grain ne sont pas vérifiées. La solution dans le cas présent consiste à trier les arcs lexicographiquement et à faire une somme préfixée.

Le placement des données est très important dans un algorithme CGM. Aussi, le traitement parallèle efficace d'un problème a de grandes chances de nécessiter au moins un tri des données du problème. Les algorithmes parallèles linéaires sont rares. Il n'est pas étonnant qu'il faille payer le prix lorsque les données sont distribuées. On peut toutefois obtenir un facteur $\log p$ au lieu de $\log n$, avec le tri du géomètre ou les algorithmes de tri probabilistes [68]. Qu'en est-il des tris implantés par les constructeurs de machines parallèles ?

Les graphes qui ont de grandes disparités dans les degrés sont plus difficiles à traiter que ceux qui font preuve de régularité. On pourrait alors chercher dans cette optique des transformation de graphes qui permettent de rendre le degré plus régulier. Il existe par exemple une technique PRAM pour les algorithmes de

« contraction d'arbres » consistant à rendre le degré constant en rajoutant à peu près autant de sommets qu'il y a d'arcs [86, 60]. Cette technique s'étend aisément aux graphes. Peut-on trouver des techniques similaires en CGM?

Conclusion

Nous avons transformé quelques algorithmes PRAM en CGM. Le tri et les calculs préfixés sont encore des outils de base. Le calcul des composantes connexes paraît en revanche assez lourd. Le modèle CGM pose de plus le problème de la distribution des données. Si la solution suit naturellement l'algorithme PRAM dans certains cas (comme la reconnaissance des ordres N-free), on est confronté à d'importants problèmes d'équilibrage de charge entre les différents processeurs dans d'autres (comme la réponse à des requêtes en petit nombre par rapport à la taille de la base de données, ou le calcul des classes d'implication d'un graphe). Il paraît parfois difficile d'équilibrer à la fois la distribution des données et la distribution des calculs.

Chapitre 3

Bornes d'un réseau de téléphones mobiles : une machine parallèle à communications restreintes

Nous allons aborder dans ce chapitre un problème issu du monde industriel. Au cours d'une collaboration avec NORTEL MATRA CELLULAR, Jean-Louis DORNSTETTER a eu la gentillesse de nous informer de quelques problèmes algorithmiques rencontrés en téléphonie cellulaire. Je vois deux bénéfices dans une telle approche : le premier est de tester l'utilité des techniques générales qui sont développées en algorithmique, le deuxième est de s'informer des problèmes théoriques auxquels on peut être confronté en pratique.

Le problème abordé ici consiste à trouver un algorithme distribué pour synchroniser les horloges des bornes d'un réseau de téléphonie GSM. La seule information dont on dispose pour cela est la détection fortuite de la différence d'heure entre deux bornes lorsqu'un téléphone cellulaire passe entre elles. Seules ces deux bornes peuvent être informées de ce décalage. La principale difficulté du problème consiste à s'interdire l'envoi de messages entre les bornes. Le terme « distribué » est donc à prendre avec des pincettes puisque l'on est confronté ici à une sorte de machine distribuée où les communications sont de nature fixée et totalement incontrôlables. Il y a de plus une autre difficulté quant à la conception d'algorithmes dans un tel système : les différents programmes vont tourner de manière totalement asynchrone sur les différents nœuds du réseau, il n'y a pas moyen de synchroniser l'exécution d'une phase du programme entre les différentes bornes. « Synchroniser » est ici utilisé dans le contexte du parallélisme ; tous les algorithmes parallèles classique utilisent de la synchronisation, et la plupart reposent même sur des barrières de synchronisation, c'est-à-dire un top à partir du quel tous les processeurs commencent en même temps une phase de calcul.

En pratique, on peut évidemment alléger ces deux interdictions mais dans une faible mesure : il existe un réseau de communication entre les bornes mais il est déjà très chargé et on peut obtenir une synchronisation approximative des calculs

en envoyant un top du central vers les bornes par ce réseau de communication ou en lançant des phases de calcul à une même heure de la journée[1]. Le but de cette étude était de faire la collecte des idées algorithmiques qui peuvent malgré tout s'appliquer dans un contexte aussi restreint. Le paragraphe 3.2 rassemble les idées que nous avons pu piocher dans la littérature ou imaginer.

La présentation du problème de synchronisation des bornes d'un réseau GSM est assez longue car les contraintes sont nombreuses. Il va de soi qu'aucun algorithme existant dans la littérature ne permet de résoudre un problème aussi spécifique. D'autre part, on ne peut proposer une solution unique car le seul moyen de prouver qu'un algorithme est meilleur qu'un autre est de les implanter tous les deux et de constater lequel donne de meilleurs résultats dans la réalité. Il y a tout de même un pas intermédiaire qui consiste à simuler les différents algorithmes dont on a l'idée. Nous avons donc suivi cette démarche, le paragraphe 3.3 de ce chapitre est consacré à la présentation des résultats des simulations que nous avons effectuées.

3.1 Présentation du problème

Réseau de téléphones mobiles

Un réseau de téléphones mobiles GSM est composé, outre les téléphones mobiles eux-mêmes que l'on appelle *mobiles*, de *bornes* fixes («*Base Tranceiver Stations* »). Les bornes sont reliées physiquement au réseau téléphonique commuté classique. Les communications sont assurées par liaison radio du mobile à une borne dans sa proximité puis par liaison filaire de la borne au réseau téléphonique standard. Les bornes sont reliées entre elles par un réseau en arbre, elles sont reliées à des stations de contrôle qui sont elles-mêmes reliées à des standards téléphoniques d'une part et à une station de maintenance d'autre part. Voir la figure 3.1.

Lors d'une communication, le mobile devient esclave d'une des bornes alentours. Il communique par paquets avec la borne. Ces paquets sont envoyés périodiquement à une fréquence donnée par une horloge (un paquet dure un huitième de la période). Le mobile cale son horloge sur celle de la borne pour être en phase avec celle-ci. Chaque paquet comporte une partie chiffrant la communication elle-même, et une partie standard toujours identique servant à différentes optimisations de traitement du signal : réglage de l'antenne, recherche de la phase, traitement des échos et des interférences.

Chaque borne possède sa propre horloge indépendante. Le problème consiste à mettre les horloges des bornes voisines en phase. Il y a deux raisons à cela. Quand un mobile se déplace, il est pris en charge par différentes bornes successives.

1. Les horloges que l'on veut synchroniser ont une période trop courte pour tenir ce rôle, mais tout ordinateur est en général muni d'une horloge classique.

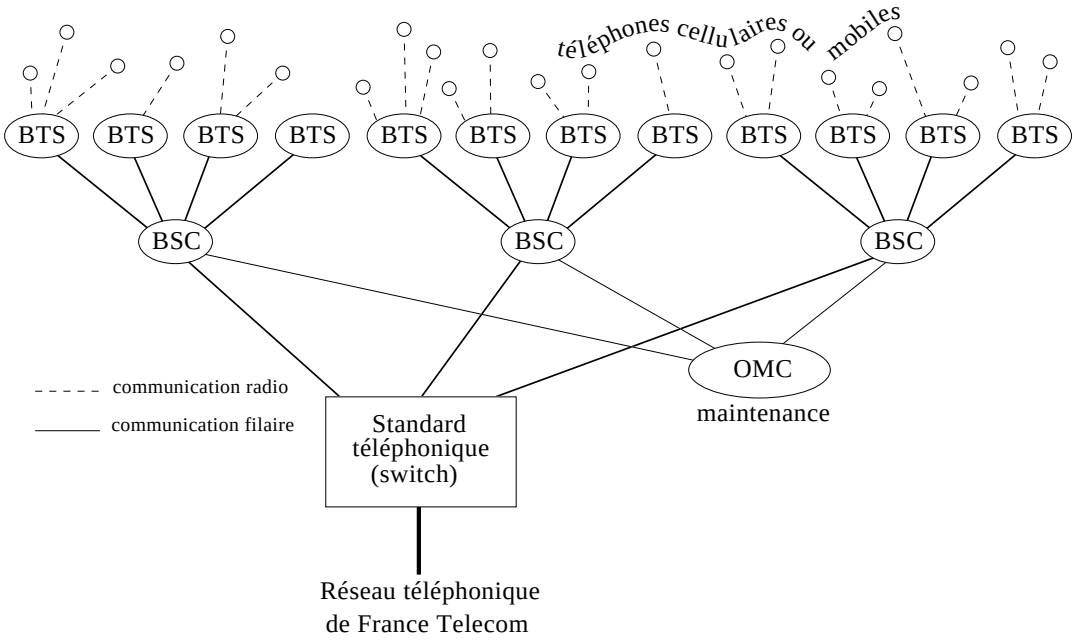


FIG. 3.1 – *Un réseau de téléphonie* GSM.

Lorsqu'il passe de l'une à l'autre, il doit émettre des paquets plus courts pour ne pas perturber les communications de la borne sur laquelle il arrive jusqu'à ce que celle-ci lui communique la phase sur laquelle il doit se recaler. Cela insère un petit blanc dans la communication, désagréable pour l'utilisateur. Si les deux bornes sont presque en phase, une technique appelée «*hand-over* » permet de raccourcir la durée du blanc. D'un autre côté, il y a beaucoup de mobiles qui se brouillent les uns les autres. Deux mobiles proches mais communiquant avec deux bornes différentes peuvent se brouiller de manière destructive quand leurs phases sont décalées (voir la figure 3.2).

Actuellement les réseaux GSM fonctionnent malgré ces problèmes, mais l'augmentation du nombre de mobiles en circulation pourrait compromettre le bon fonctionnement d'un tel réseau. Il y a deux approches complémentaires pour résoudre ce problème en jouant sur les trois principaux paramètres qu'un administrateur de réseau peut contrôler. La première option consiste à minimiser les interférences en optimisant la puissance d'émission demandée à chaque mobile et le choix de la borne qui s'occupe de lui. Nous nous intéresserons ici à la deuxième approche : synchroniser les horloges de bornes pouvant interférer. Ceci est possible car les bornes peuvent modifier très lentement leur phase[2].

Le moyen le plus simple d'effectuer cela est technologique : munir chaque borne d'un système de synchronisation. La seule source économique de synchronisation

2. Des changements brusques sont interdits car ils feraient sauter toutes les communications en cours et sont d'ailleurs soumis à des règles précises par la norme CGM.

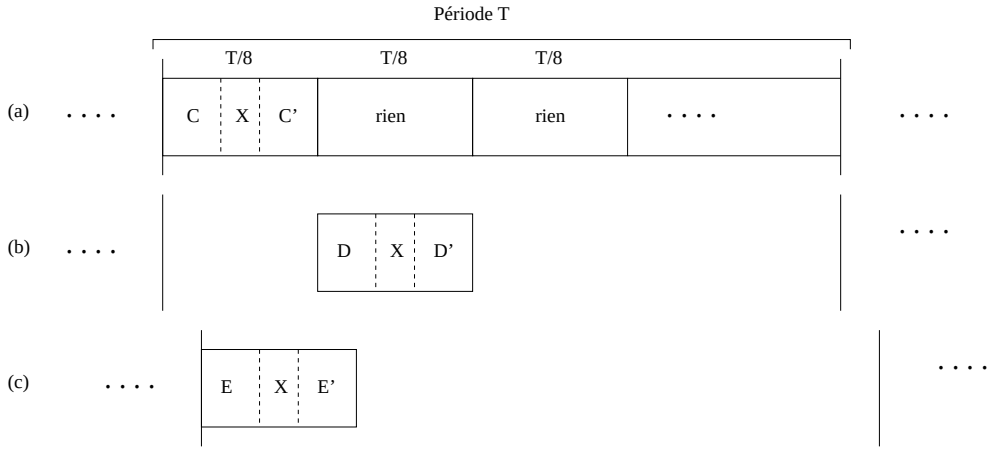


FIG. 3.2 – *(a) Schéma du signal émis par un mobile. La partie X est fixe et standard et sert à mesurer les interférences qui sont venues perturber la transmission de ce paquet pour tenter de les corriger. (b) Signal d'un autre mobile communicant avec la même borne. (c) Signal émit par un mobile communicant avec une autre borne dont la phase est décalée par rapport à la première borne. Se mobile vient brouiller les deux autres signaux. Ces interférences seront prises en compte pour le mobile a puisque la partie standard X est aussi brouillée, tandis qu'elles ne le seront pas pour le mobile b (sa partie X n'est pas brouillée). La correction sur le paquet du mobile a va dégrader la partie non brouillée et la partie brouillée du paquet du mobile b ne sera pas corrigée en conséquence. Il en découle une dégradation de la qualité des communications qui n'a pas lieu lorsque le brouilleur est à peu près en phase avec la borne.*

assez précise à l'heure actuelle se trouve être le système GPS mis en œuvre par l'armée américaine. En effet, seules les horloges atomiques transportées par les différents satellites en orbite basse de ce système permettent la précision nécessaire (de l'ordre de 10^{-7} secondes). Le prix actuel d'un GPS est tel que cette solution est tout-à-fait acceptable économiquement, mais d'un point de vue politique, il est délicat de faire reposer le fonctionnement d'un réseau de communication sur le bon vouloir de l'armée américaine. Le plus gros inconvénient de cette méthode est qu'il faut rajouter une antenne pour chaque borne, ce qui pose de nombreux problèmes d'autorisations.

Une particularité de la norme GSM permet d'envisager une solution logicielle meilleur marché et ne présentant pas les inconvénients politiques de l'option précédente. Les bornes sont équipées d'un système spécial permettant d'écouter les signaux provenant de mobiles communiquant avec une borne voisine. Il se trouve donc que chaque borne connaît de temps en temps son *décalage* de phase avec celles qui lui sont *voisines*, c'est-à-dire qui peuvent interférer avec elle. Cette information est complètement incontrôlable et n'est en rien prévisible, elle dé-

pend du passage fortuit de mobile entre les bornes. Remarquons tout de même qu'il est important de synchroniser les bornes telles que de nombreux mobiles passent entre elles en créant des interférences gênantes. Les interférences les plus gênantes sont donc celles qui sont le plus souvent détectées. On peut donc définir un graphe dont les sommets sont les bornes et dont les arêtes relient les bornes qui interfèrent assez souvent pour que leur décalage soit détecté de temps en temps. Ce graphe est inconnu, il dépend des obstacles présents entre les bornes. Il n'est pas planaire à priori. La seule propriété que l'on en connaisse est que des bornes géographiquement éloignées sont moins susceptibles d'interférer. Ceci n'est pas une généralité car une borne placée en haut de la tour Montparnasse par exemple peut interférer avec toutes les autres bornes de Paris. Le nombre de telles bornes est faible à priori et les sommets du graphe ont probablement un degré assez faible en règle générale.

Les bornes peuvent communiquer par le réseau en arbre les reliant à la station de maintenance. Cependant, ce réseau est largement utilisé pour d'autres besoins et il est préférable d'en limiter au maximum l'utilisation. La solution doit donc être un algorithme distribué utilisant au maximum les informations locales reçues par chaque borne.

Modélisation du problème

Le graphe sur les bornes constitue un premier pas dans la modélisation mais reste un peu flou car les arêtes qui correspondent à la détection d'un décalage entre deux bornes dépendent du passage fortuit d'un mobile entre les bornes. Précisons cela en discrétisant le temps (aux heures de trafic intense, une borne reçoit typiquement des décalages à quelques secondes d'intervalle, l'unité de temps choisie peut par exemple être la minute). On peut alors considérer la probabilité $p_{i,j}$ pour que la borne i reçoive son décalage avec la borne j à chaque instant. On définit ainsi une *matrice de probabilité* qui modélise le réseau GSM.

La disposition géographique des bornes étant fixe, les bornes qui interfèrent avec une certaine régularité avec une borne donnée sont toujours les mêmes. Sur le nombre, les déplacements des mobiles étant toujours à peu près distribués de la même manière, on peut supposer cette matrice constante (quitte à supposer que le temps ne s'écoule pas linéairement puisque le trafic est certainement moins important la nuit[3]). Cette matrice est symétrique puisqu'un décalage entre deux bornes est détecté lorsqu'un mobile peut être pris en charge par les deux bornes. En fixant un seuil de probabilité au delà duquel deux bornes interfèrent suffisamment pour qu'il soit nécessaire et aussi possible de les synchroniser on obtient un graphe non orienté au sens usuel du terme : deux bornes sont reliées si la probabilité correspondante est supérieure au seuil.

3. Cette simplification peut nécessiter des adaptations de l'algorithme comme recaler les horloges à une vitesse qui dépend du nombre de décalages reçus par unité de temps.

Le problème de synchroniser les bornes peut être vu comme un calcul de composantes connexes de ce graphe. En effet, deux bornes reliées par un chemin seront synchronisées si chaque borne est synchronisée avec ses voisines, et deux bornes interférant trop peu fréquemment n'ont pas besoin d'être synchronisées. Le problème se formule dans cette optique comme la recherche d'un algorithme de composantes connexes dans ce modèle de calcul distribué où les communications sont très restreintes.

3.2 Algorithmes de synchronisation

Algorithme du gradient

La solution classique est de se ramener à un calcul de minimum. Il est utile de faire l'analogie avec la physique tant pour stimuler l'intuition que pour employer des termes plus imagés : le problème consiste à minimiser « une *fonction d'énergie* ». Dans un système distribué comme celui considéré ici, on essayera plutôt d'exprimer cette fonction d'énergie comme la somme de fonctions d'énergie *locales*. Une fonction d'énergie locale toute trouvée ici est la somme des carrés des décalages d'une borne avec ses voisines.

Notons θ_i l'heure de la borne i et Δ_{ij} le décalage entre les bornes i et j. Précisons que seuls les $\Delta_{ij} = \theta_j - \theta_i$ sont mesurés, les θ_i ne sont pas mesurables (il n'y a pas d'heure absolue). D'autre part, les θ_i sont des heures modulo la période T de l'horloge[4]. C'est cet aspect modulaire des variables qui rend le problème difficile. Ramenons par convention la valeur de Δ_{ij} entre $-T/2$ et $T/2$ (c'est une valeur modulo T).

$$\Delta_{ij} = (\theta_j - \theta_i) \bmod T, \quad -\frac{T}{2} < \Delta_{ij} \leq \frac{T}{2}.$$

Une borne peut se recaler sur une autre de deux manières possibles : soit en avançant, soit en retardant. La manière la plus rapide demande un temps $v_r\,|\Delta_{ij}|$ (où v_r est la vitesse à laquelle les bornes peuvent se recaler), l'autre manière demande $v_r(|\Delta_{ij} - T|)$ (voir la figure 3.3). Il y a des situations où le seul moyen de synchroniser toutes les horloges consiste à recaler une des horloges sur une autre en prenant le chemin le plus long (voir la figure 3.4). Le problème consiste à décider pour chaque borne si elle doit se recaler dans un sens ou dans l'autre.

Malgré tout, lorsque les bornes sont à peu près synchronisées (par exemple tous les Δ_{ij} sont entre 0 et $3T/4$), on peut utiliser la solution classique consistant à minimiser une fonction d'énergie. Définissons donc une fonction d'énergie globale

4. Cette période est la même pour toutes les bornes. Il faut avoir en tête l'analogie d'une horloge traditionnelle (pour laquelle $T = 12$ heures) ou du cercle trigonométrique (pour lequel $T = 2\pi$).

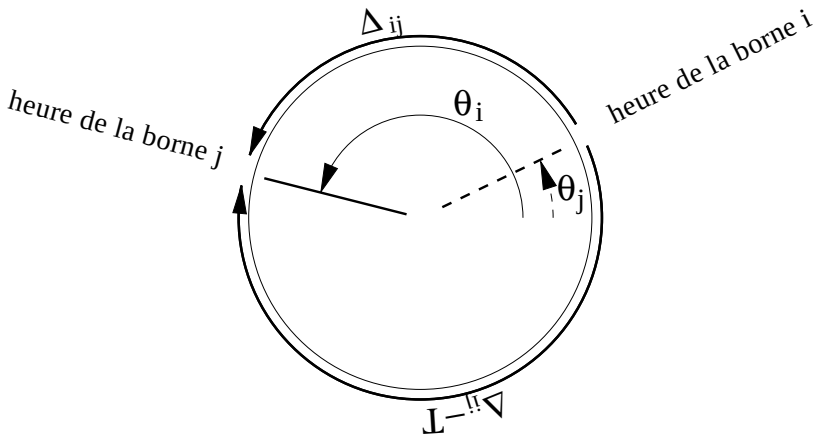


FIG. 3.3 – *Les deux valeurs possibles pour le décalage entre deux bornes. La borne i peut se recaler sur la borne j soit en retardant de $|\Delta_{ij} - T|$, soit en avançant de $|\Delta_{ij}|$.*

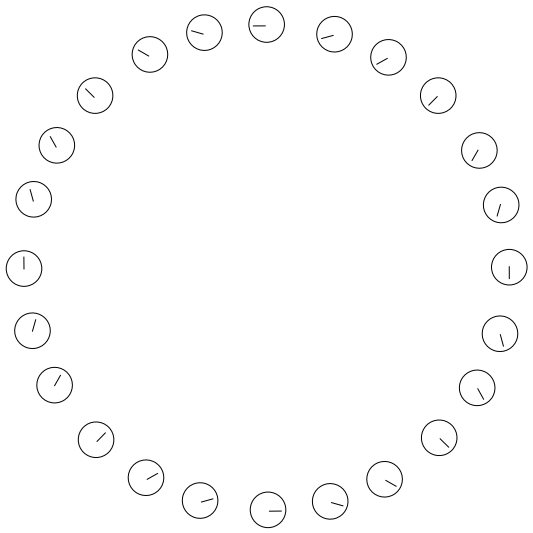


FIG. 3.4 – *Situation de blocage : pour synchroniser les horloges, il faut que l'une d'entre elles se recale sur une de ses voisines en passant par l'arc de cercle le plus long, c'est-à-dire en faisant augmenter la valeur de $|\Delta_{ij}|$ (en considérant que c'est la borne i qui se recale vers la borne j de cette manière).*

F et des fonctions d'énergie locales f_i ainsi :

$$F(\theta_1, \ldots, \theta_n) = \sum_{1 \leq i,j \leq n} \Delta_{ij}^2 = \sum_{0 \leq i < n} f_i(\theta_1, \ldots, \theta_n)$$

$$f_i(\theta_1, \ldots, \theta_n) = \sum_{1 \leq j \leq n} \Delta_{ij}^2.$$

La fonction F définit une surface dans R^{n+1} dont il s'agit de trouver un point minimum. L'algorithme le plus simple pour trouver un minimum d'une surface consiste à se déplacer le long de la surface en suivant la ligne de plus grande pente qui est donnée par le gradient ∇F défini par :

$$\nabla F = \left(\frac{\partial F}{\partial \theta_1}, \ldots, \frac{\partial F}{\partial \theta_{n-1}} \right).$$

$$\begin{aligned} \frac{\partial F}{\partial \theta_i} &= 2 \frac{\partial}{\partial \theta_i} \left[\sum_{1 \leq j \leq n} (\theta_j - \theta_i)^2 \right] \\ &= 4 \sum_{1 \leq j \leq n} (\theta_j - \theta_i) \\ &= 4 \sum_{1 \leq j \leq n} \Delta_{ij} \end{aligned}$$

L'algorithme du gradient consiste à répéter l'opération

$$(\theta_1, \ldots, \theta_n) \leftarrow (\theta_1, \ldots, \theta_n) - \varepsilon \nabla F(\theta_1, \ldots, \theta_n).$$

ε est une constante à ajuster de sorte que l'algorithme n'oscille pas et qu'il converge assez vite. Cet algorithme se prête tout à fait à la distribution : chaque borne i peut calculer localement $\frac{\partial F}{\partial \theta_i}$ et recaler θ_i en fonction de cette valeur.

Dans le problème de synchronisation présent, les bornes vont se recaler dans un sens ou dans l'autre selon le signe de la « moyenne » $\sum_{1 \leq j \leq n} \epsilon_{ij} \Delta_{ij}$. Remarquons que le gradient de F n'est pas continu : lorsque deux horloges sont en opposition de phase, Δ_{ij} peut passer de T à $-T$ et réciproquement ; cependant, on peut arguer que l'algorithme conduit à s'éloigner de ces positions et qu'il ne peut pas osciller autour d'une telle position.

Une idée de Jean-Louis DORNSTETTER, basée sur une analogie avec les verres de spin [23], permet de choisir une fonction d'énergie plus lisse pour remédier à ce problème. L'idée est de ne pas recaler deux bornes en suivant toujours le plus court chemin mais en restant fidèle à la direction par laquelle on a commencé à se recaler. Il suffit pour cela de calculer les Δ_{ij} modulo $2T$ de sorte que $-T < \Delta_{ij} \leq T$. Cela ne pose pas de problème en suivant l'évolution des décalages. Si jamais une borne détecte un décalage avec une voisine de $T/2 - \delta$ puis de $-T + \delta'$, on posera artificiellement $\Delta_{ij} = T + \delta'$. De même, si le décalage détecté passe de $-T + \delta'$ à $T - \delta$, on posera $\Delta_{ij} = -T - \delta$. La fonction d'énergie est alors modifiée de sorte que le calcul de la moyenne des décalages dans le calcul du gradient

devienne

$$\sum_{1\leq j\leq n} g(\Delta_{ij}) \quad \text{où} \quad g(\Delta_{ij}) = \begin{cases} \Delta_{ij} & \text{si } -\frac{T}{2} \leq \Delta_{ij} \leq \frac{T}{2} \\ T - \Delta_{ij} & \text{si } \frac{T}{2} \leq \Delta_{ij} \leq T \\ -T - \Delta_{ij} & \text{si } -T \leq \Delta_{ij} \leq -\frac{T}{2} \end{cases}$$

L'analogie avec les verres de spin conduit à poser $g(\Delta_{ij}) = \frac{T}{2}\sin(\frac{\pi}{T}\Delta_{ij})$, ce qui lisse complètement la fonction d'énergie mais augmente considérablement les calculs (surtout dans les simulations). La figure 3.5 illustre la forme des différentes fonctions d'énergie et les fonctions g associées.

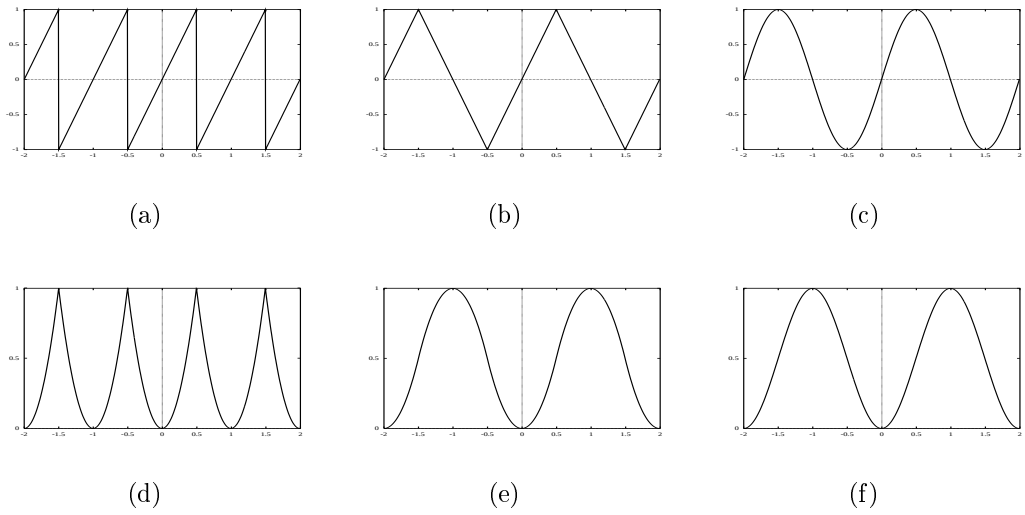
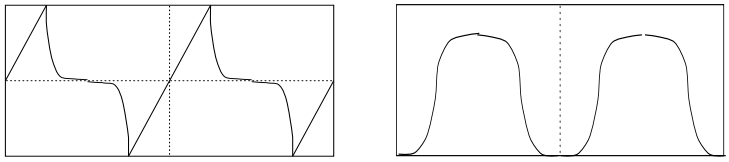


FIG. 3.5 – *(d) Fonction d'énergie simple, de gradient (a) discontinu. (e) Fonction d'énergie lissée de gradient (b) continu. (f) Fonction d'énergie $\mathcal{C}^\infty$ de gradient (c) sinusoïdal.*

Cette méthode pourrait cependant être dangereuse : deux bornes pourraient se « courir » l'une après l'autre dans le cas où elles ne seraient pas d'accord sur la valeur de leur décalage ; on pourrait avoir $\Delta_{ij} = -\Delta_{ji} + T$ (ce qui ne contredit pas l'égalité de Δ_{ij} et $-\Delta_{ji}$ modulo T), au quel cas les deux horloges se décaleraient dans le même sens sans pouvoir se rattraper l'une l'autre. Dans le cas du réseau GSM, ce danger n'existe pas car les horloges ont en réalité une période supérieure à la période à laquelle on veut qu'elles soient synchronisées (les décalages sont détectés modulo $8T$). Même si ce n'était pas le cas, on peut imaginer une méthode permettant à une borne de rattraper l'autre. Si une borne recale son horloge à une vitesse proportionnelle au gradient, tout en gardant le gradient continu, on peut le rendre plus grand pour la borne qui se recale en suivant le plus court chemin que pour la borne qui se recale en suivant le plus long chemin (voir la figure 3.6). Un autre défaut de la méthode de lissage de la fonction d'énergie est

que le nombre de minima absolus dans une région bornée de R^n donnée est divisé par 2^n. Je ne sais pas dans quelle mesure cela est gênant.



(a) (b)

FIG. 3.6 – *Un lissage de la fonction d'énergie (b) (de gradient (a)) tolérant au désaccord de deux bornes sur la valeur de leur décalage modulo* $2T$.

Malheureusement, la méthode du gradient ne permet pas toujours d'atteindre un minimum. Dans le cas où les bornes dans leur ensemble sont proches de la synchronisation (ce qui est peu probable quand il y a beaucoup de bornes), cette méthode permet effectivement de trouver un minimum d'énergie ($F = 0$). Mais en partant d'une position quelconque, il peut y avoir des blocages : l'algorithme peut rester bloqué dans un minimum local tel que la position illustrée par la figure 3.4.

Cette méthode implique que chaque borne doit retenir un nombre pour chacune de ses voisines. Cela est possible, mais les ressources étant très limitées sur chaque borne, mentionnons une idée qui permet de l'éviter. On peut remarquer que les horloges se recalent très lentement (à cause des limitations imposées par la norme GSM). On peut faire un calcul empirique de la moyenne des décalages en recalant systématiquement l'horloge d'une borne vers celle d'une voisine dès qu'un décalage est mesuré avec celle-ci. Lorsque les bornes commencent à se synchroniser et les décalages commencent à devenir faibles, il faut bien sûr faire baisser cette vitesse de recalage pour ne pas osciller autour de la position de synchronisation. Cette méthode a de plus l'avantage de pondérer la moyenne avec la fréquence de détection du décalage avec chaque voisine. Une borne se recalera donc préférentiellement vers la voisine avec laquelle elle a le plus d'interférences. Si l'on veut éviter cette pondération, il suffit de rendre artificiellement toutes les fréquences de détection égales à la plus petite en ne tenant pas compte du surplus de détections. L'algorithme 3.1 donne les détails de cette méthode qui permet de synchroniser de manière stable les bornes lorsqu'elles sont sur la bonne voie. Le reste de la section s'intéresse plutôt au cas où les bornes sont loin de la synchronisation. On peut imaginer une stratégie générale où une borne applique un algorithme quand ses voisines sont dispersées et l'algorithme du gradient quand elles sont plutôt groupées.

Le facteur $1/2$ dans $c_2 \left|g(\Delta_{ij})\right| /2$ indique qu'une borne ne doit pas se recaler d'une valeur trop importante pour éviter que le système n'oscille : deux bornes

Algorithme 3.1 Gradient pour la borne i

Répéter

Dès qu'un décalage Δ_{ij} est détecté, effectuer :

Recaler l'horloge à la vitesse $c_1\, v_r\, g(\Delta_{ij})$ (dans le sens donné par le signe de la vitesse) d'au plus $c_2\, |g(\Delta_{ij})|\, /2$.

décalées de Δ doivent se recaler l'une de $\Delta/2$, l'autre de $-\Delta/2$ pour se synchroniser. N'importe quelle valeur inférieure strictement à 1 convient. Pour une optimisation plus fine, il faudrait certainement prendre des fonctions plus compliquées que de simples multiplications par les coefficients c_1 et c_2 (des fonctions affines par morceau par exemple).

L'algorithme 3.1 est sûrement le plus simple que l'on puisse imaginer. Il pourrait y avoir des problèmes de stabilité lorsque les mesures de décalage sont bruitées : une erreur de mesure peut amener une borne à recaler son horloge dans la mauvaise direction jusqu'à ce qu'elle reçoive un décalage moins bruité. La vitesse de recalage étant très faible par rapport à la fréquence de détection de décalage, on peut imaginer que cela n'aura pas une grande incidence sur l'algorithme. Cela peut toutefois être gênant lorsque les bornes sont synchronisées ou lorsque la fréquence de détection de décalage tombe (la nuit par exemple). On préfèrera alors la variante 3.2 un peu plus compliquée où l'on calcule effectivement la dérivée partielle $\frac{\partial F}{\partial \theta_i}$ de la fonction d'énergie.

Algorithme 3.2 Gradient pour la borne i

Données : Les décalages Δ_{ij} pour chaque voisine j de la borne i.

Début

Tenir les Δ_{ij} à jour à chaque décalage détecté.

Effectuer à intervalle de temps régulier :

$G \leftarrow \sum_{1 \leq j \leq n} g(\Delta_{ij})$

Recaler l'horloge à la vitesse $c_1\, v_r\, G$ d'au plus $c_2\, |G|\, /2$.

Fin

Quand un nouveau décalage est détecté on peut même faire la moyenne avec l'ancienne valeur pour rendre l'algorithme encore plus stable face aux erreurs de détection. Les variations sont bien sûr infinies, mon but est de rassembler ici les principales idées que nous avons imaginées.

Le recuit simulé

Le recuit simulé est un algorithme probabiliste générique permettant de résoudre en théorie tous les problèmes modélisables par la minimalisation d'une fonction d'« énergie » [48, 1, 72]. L'idée s'exprime encore naturellement avec

l'analogie à la physique. La fonction F dont il faut trouver un minimum est considérée comme un *potentiel* dont on cherche les puits les plus profonds. On considère alors des particules soumises à ce potentiel et à une *agitation thermique*. La distribution des particules, selon les travaux de BOLTZMANN, est alors donnée par

$$Prob\left[(\theta_1, \ldots, \theta_n) = \Theta\right] = e^{-\frac{F(\Theta)}{k\tau}}$$

où τ est la température du système. Ici, le réseau entier de bornes est considéré comme une seule particule se déplaçant dans R^n. Toute suite $(\theta_1, \ldots, \theta_n) \in R^n$ forme un état possible de la particule. A une température τ une particule peut passer d'un état Θ à un état Ξ avec une probabilité

$$Prob\left[\Theta \longrightarrow \Xi\right] = e^{-\frac{F(\Xi)-F(\Theta)}{k\tau}}$$

Si la température est infinie, tous les états sont équiprobables, une particule peut sauter n'importe quelle barrière de potentiel pour passer d'un état à un autre. Si la température est nulle, toutes les particules sont dans des puits de potentiel, c'est-à-dire des minima locaux de F, aucune agitation thermique ne leur permet d'en sortir. En revanche, si l'on fait baisser doucement la température de l'infini vers zéro, toutes les particules se retrouveront dans les minima absolus de F (ces minima sont équiprobables). On peut donner une explication physique intuitive à cela : quand la température est assez élevée, une particule passe de minimum local en minimum local[5] ; si une particule hésite entre deux minima, il y a une température qui lui permettra de passer du moins profond au plus profond, mais pas du plus profond au moins profond, une fois piégée dans le plus profond, elle ne peut plus en sortir (voir la figure 3.7). Il existe une justification mathématique de cette propriété du refroidissement d'un système sous les hypothèses qui suivent [36, 1].

La température doit baisser très lentement (trop lentement pour donner un algorithme qui termine, mais il existe des stratégies pour refroidir plus rapidement, voir un peu plus loin). D'autre part, une particule doit pouvoir passer de tout état à n'importe quel autre avec une probabilité non nulle en un temps fini. Cela peut se modéliser mathématiquement par l'existence d'une *matrice d'exploration* c'est-à-dire une matrice M symétrique et irréductible où $M[a, b]$ est la probabilité de passer de l'état Θ à l'état Ξ. Cela veut essentiellement dire que $M[\Theta, \Xi] = M[\Theta, \Xi]$ et qu'une particule peut « passer » d'un état Θ^1 à un état Ξ en passant éventuellement par des états intermédiaires $\Theta^2, \ldots, \Theta^b = \Xi$ tels que $M[\Theta_a, \Theta_{a+1}] > 0$ pour tout $1 \leq a < b$. Ceci est la définition usuelle, M est aussi un graphe non orienté connexe sur l'ensemble des états où deux états sont reliés si la probabilité de passer de l'un à l'autre directement est non nulle. En pratique [2], les arêtes ne sont pas valuées, toutes les arêtes partant d'un sommet sont posées équiprobables.

5. Si la température n'est pas infinie, la probabilité d'être dans un puits de potentiel est d'autant plus forte que le puits est profond.

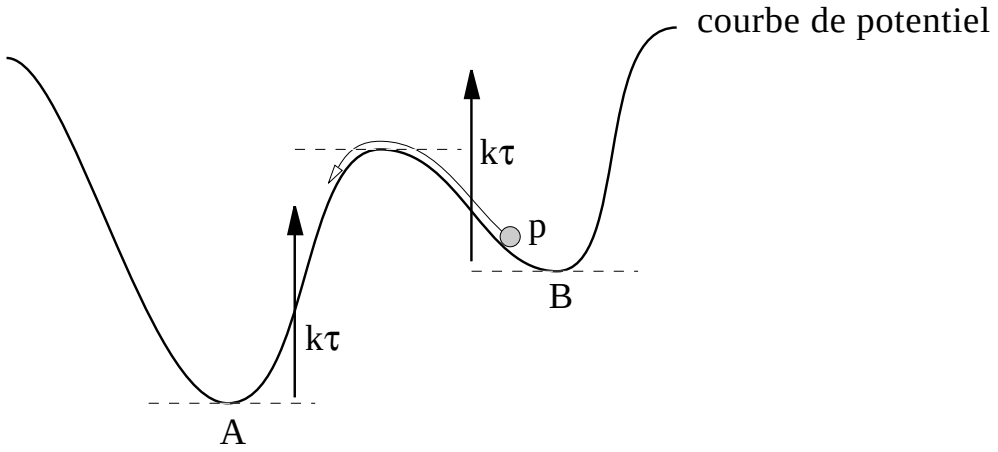


FIG. 3.7 – *Une particule d'énergie thermique $k\tau$ peut passer du puits B dans le puits A, mais pas l'inverse.*

Du point de vue algorithmique, cette matrice d'exploration permet de se déplacer aléatoirement d'un état à un autre. Le choix d'une distribution de BOLTZMANN ne provient pas uniquement de la pertinence de l'analogie avec ce modèle physique, elle est aussi facile à simuler grâce à l'algorithme de Métropolis [48]. La matrice d'exploration permet en effet de simuler le déplacement d'une particule soumise au potentiel F et à une agitation thermique donnant lieu à une distribution de BOLTZMANN.

L'analogie physique est très satisfaisante intuitivement car la décroissance de la température consiste à faire diminuer l'entropie jusqu'au zéro absolu, c'est-à-dire un système totalement ordonné, ou encore un cristal, ou encore un ensemble de bornes synchronisées. C'est probablement aller un peu loin que de justifier un algorithme informatique par les principes de la thermodynamique mais il se trouve que le recuit simulé permet de résoudre de nombreux problèmes de ce type. L'algorithme du recuit simulé consiste à calculer une suite d'états $\Theta^0, \ldots, \Theta^a, \ldots$ à partir d'un état initial Θ^0 et d'une suite de températures $\tau^0 = \infty, \tau^1, \ldots$ qui tendent vers 0 à l'infini en répétant la procédure donnée par l'algorithme 3.3.

La suite $(\Theta^a)_{a \geq 0}$ est une chaîne de MARKOV dont la matrice de transition varie dans le temps. Dans l'algorithme général du recuit simulé, la partie critique se trouve dans la fonction de refroidissement *Refroidir*. Le premier résultat sur la convergence du recuit simule [36] montre que

$$\lim_{a \to \infty} Prob\,[\Theta^a \text{ soit un minimum absolu}] = 1$$
$$\text{quand } \lim_{a \to \infty} \tau^a \log a \geq R \text{ avec } R \geq 0 \text{ assez grand.}$$

Ceci est un résultat mathématique, qui a une traduction exploitable en informa-

Algorithme 3.3 [**36**] Recuit simulé

Répéter
  Choisir aléatoirement un état Ξ voisin de Θ^a.
  Si $F(\Xi) \leq F(\Theta^a)$ **Alors**
    $\Theta^{a+1} \leftarrow \Xi$
  Sinon
    Tirer un nombre aléatoire q entre O et 1.
    Si $q < e^{-\frac{F(\Xi)-F(\Theta^a)}{\tau^a}}$ **Alors** $\Theta^{a+1} \leftarrow \Xi$
  $\tau^{a+1} \leftarrow \mathit{Refroidir}(a)$
  $a \leftarrow a + 1$

tique :

$$\mathit{Prob}\,[\Theta^a \text{ ne soit pas un minimum absolu}] \sim \left(\frac{C}{n}\right)^r$$

où C et r sont des constantes. Il est donc possible de borner la probabilité pour que l'algorithme n'ait pas terminé au bout de a itérations. Le recuit simulé est donc un algorithme probabiliste.

Cette stratégie de refroidissement est bien trop lente pour être utile. Cependant, une stratégie classique consiste à prendre $\tau^a = \tau^0 B^a$ où B est une constante proche de 1 (typiquement $B = 0,99$). Il existe des résultats plus faibles sur la convergence de cette stratégie [10, 1, 72, 2][6]. Remarquons que la constante C est d'autant plus grande que l'espace des états est grand.

Cette méthode peut être adaptée au problème de la synchronisation des bornes d'un réseau GSM de la manière suivante. Chaque borne contrôle une des coordonnées de la particule. La matrice de transition n'est pas difficile à imaginer : une borne i peut faire passer le système dans l'état $(\theta_1, \ldots, \theta_i + \varepsilon, \ldots, \theta_n)$ ou dans l'état $(\theta_1, \ldots, \theta_i - \varepsilon, \ldots, \theta_n)$. Le principal problème est qu'une borne ne peut pas calculer la fonction d'énergie qui est globale. En s'inspirant de l'idée de l'algorithme du gradient, on peut cependant remédier à cela. Pour $\Xi = \Theta + (\varepsilon_1, \ldots, \varepsilon_n)$, on peut utiliser l'approximation

$$F(\Xi) - F(\Theta) \simeq \nabla F \odot (\varepsilon_1, \ldots, \varepsilon_n)$$

où $\odot$ désigne le produit scalaire. Une borne peut donc calculer localement

$$F(\theta_1, \ldots, \theta_i \pm \varepsilon, \ldots, \theta_n) - F(\theta_1, \ldots, \theta_n) = \pm\varepsilon \times 4 \sum_{1 \leq j \leq n} g(\Delta_{ij}).$$

Il est préférable à priori de choisir une version dérivable en tout point de la fonction d'énergie. Le recuit simulé apparaît alors comme une modification de la méthode du gradient. L'algorithme 3.4 donne les détails pour la borne i.

6. Il existe même des algorithmes parallèles de recuit simulé où plusieurs recherches indé-

Algorithme 3.4 Recuit simulé pour la borne i

Données : Les décalages Δ_{ij} pour chaque voisine j de la borne i.

Début

Tenir les Δ_{ij} à jour à chaque décalage détecté.

Effectuer à intervalle de temps régulier :

$G \leftarrow \sum_{1 \leq j \leq n} g(\Delta_{ij})$

Poser avec probabilité $1/2$: $\alpha \leftarrow 1$ ou bien $\alpha \leftarrow -1$.

Si $\alpha G \geq 0$ **Alors**

Recaler l'horloge à la vitesse $c_1 \, v_r \, G$ d'au plus $\min(c_2 \, |G| \, /2, \varepsilon)$.

Sinon

Tirer un nombre aléatoire q entre O et 1.

Si $q < e^{\frac{G \alpha \varepsilon}{\tau^a}}$ **Alors**

Recaler l'horloge à la vitesse $c_1 \, v_r \, \alpha G$ d'au plus $\min(c_2 \, |G| \, /2, \varepsilon)$.

$\tau^{a+1} \leftarrow \mathit{Refroidir}(a)$

$a \leftarrow a + 1$

Fin

L'intervalle de temps entre deux décisions doit permettre à l'horloge de se recaler de ε (on peut moduler de manière plus fine la vitesse de recalage en la mettant au maximum autorisé tant que les heures des voisines de la borne sont dispersées). Il faut trouver un bon compromis pour ε car il doit être petit pour que l'approximation de la fonction d'énergie soit valable, mais plus il est petit, plus le nombre d'états possibles du système devient grand et plus la convergence est lente. La solution la plus sophistiquée consiste sans doute à faire varier ε en fonction de l'amplitude des décalages.

Le plus gros problème dans cet algorithme est que les bornes ne sont pas synchronisées au sens calculatoire du terme : chacune devra avoir sa propre température et ne commence pas l'algorithme en même temps que les autres. De ce point de vue, le maximum de synchronisme entre les différentes exécutions du recuit simulé sur chaque borne serait souhaitable : si les bornes ont l'heure et la date, il serait sûrement pertinent de faire partir tous les algorithmes en même temps à une date et à une heure les plus précises possibles. Si une borne est allumée au milieu de l'algorithme, il faudra lui donner une température de l'ordre de celle des autres sans quoi elle risque de perturber fortement ses voisines.

Citons enfin une idée originale de Daniel Krob [51] qui consiste à opérer les deux phases du recuit simulé (choisir au hasard un état voisin et décider d'y passer ou pas) dans l'ordre inverse : au lieu de faire passer le système d'un état dans un autre, une borne observe un changement d'état et décide de laisser faire

pendantes sont effectuées en même temps [2] et où de temps en temps toutes les recherches repartent à partir de la meilleure d'entre elles.

ou bien d'essayer de le corriger. L'algorithme 3.5 donne les détails. La fonction d'énergie cette fois est $\sum_{1 \leq i,j \leq n} |\Delta_{ij}|$.

Algorithme 3.5 KROB

Données : Les décalages Δ_{ij} pour chaque voisine j de la borne i.

Répéter

A chaque décalage Δ_{ij} détecté, effectuer :

Comparer Δ_{ij} avec l'ancienne valeur d du décalage avec j :

Si $|\Delta_{ij}| < |d|$ **Alors**

Ne rien faire.

Sinon

Tirer un nombre aléatoire q entre O et 1.

Si q *est inférieur à* $e^{-\frac{|\Delta_{ij}|-|d|}{\tau^a}}$ **Alors**

Recaler l'horloge d'au plus $\min(c_2 |\Delta_{ij}|/2, \varepsilon)$ à la vitesse $c_1 v_r \Delta_{ij}$.

$\tau^{a+1} \leftarrow \mathit{Refroidir}(a)$

$a \leftarrow a + 1$

Composantes connexes

Les approches précédentes abordent le problème par des méthodes issues du continu : les horloges sont des variables continues, si une horloge bouge un peu, le système n'est pas profondément modifié ou perturbé. Cependant il y a deux phénomènes discrets dans ce système. Tout d'abord la commande : on peut recaler une horloge dans un sens ou bien dans l'autre. Ensuite, il y a une petite discontinuité au niveau du décalage entre deux bornes : si deux bornes sont presque en opposition de phases, leur décalage fait un saut si une borne bouge un peu au delà de la position en opposition de phases. Une approche discrète s'impose donc, au moins comme une solution complémentaire aux méthodes continues.

On peut considérer que les bornes interagissent ou n'interagissent pas, et que le problème consiste à synchroniser les composantes connexes de bornes interagissantes. Les composantes connexes sont caractérisées par l'existence d'un arbre de recouvrement. Il est intéressant d'avoir un tel arbre (de plus enraciné) pour synchroniser les bornes : chaque borne se synchronise sur son père et au bout du compte, toutes les bornes se synchronisent sur la racine de l'arbre. Cela peut poser des problèmes de stabilité aux erreurs de détection, nous aborderons ce point à la fin de la section. De là, on peut partir dans deux directions. Soit trouver un arbre couvrant « à la main », soit trouver un algorithme de calcul de composantes connexes.

Choix d'un arbre couvrant

Il y a au moins deux moyens de calculer un arbre couvrant de manière centralisée. En consultant une carte géographique des bornes, on peut certainement trouver une bonne partie des arêtes du graphes d'interaction (pas toutes) : deux bornes ont d'autant plus de chances d'interagir qu'elles sont proches, qu'il n'y a pas d'obstacle entre elles, et que beaucoup d'adeptes du téléphone cellulaire sont susceptibles de passer entre elles. Il faut ensuite désigner un père pour chaque borne et envoyer cette information à la borne, s'il y a 500 bornes, cela fait 2 kilo-octets d'informations à faire passer par le réseau inter-bornes. D'un autre côté, si l'étude d'une carte géographique ne donne pas une idée assez précise du graphe d'interaction, il est possible de calculer effectivement celui-ci : chaque borne détecte ses voisines et envoie ensuite la liste de ses voisines au central, si elle se limite à ses dix voisines les plus fréquentes, cela fait au total 12 kilo-octets d'informations à envoyer par le réseau inter-bornes (pour 500 bornes).

Si une nouvelle borne est ajoutée, il suffit qu'elle se choisisse elle-même un père (en supposant que l'on ajoute jamais deux bornes voisines en même temps). Le caractère figé de l'arbre est ennuyeux (il se peut que la construction d'un immeuble coupe une des arêtes de l'arbre, ou une borne peut tomber en panne et ses fils ne peuvent plus se synchroniser). Si l'on s'autorise plus de communications par le réseau inter-bornes, on peut recalculer l'arbre de temps en temps (toutes les nuits par exemple). Si le graphe a une connexité suffisante et que chaque borne est aussi reliée à un ancêtre de son père, on peut lui désigner un père de secours au cas où elle n'ait plus de nouvelles de son père.

Si l'on s'interdit complètement l'utilisation du réseau inter-bornes, il est toutefois possible de calculer des forêts couvrantes des composantes connexes. Il faut que chaque borne se désigne elle-même un père, la difficulté étant de ne pas construire de circuit. On peut pour cela utiliser comme dans l'algorithme parallèle un ordre total sur les bornes (donné par leurs numéros) et une borne choisit comme père une voisine de numéro plus grand. Si une borne a un numéro plus grand que ceux de ses voisines, elle n'aura pas de père et sera racine d'un arbre de la forêt couvrante. On peut aussi utiliser un ordre total sur les arêtes, leur fréquence de détection par exemple : chaque borne se recale sur sa voisine dont le décalage est détecté le plus fréquemment. Cette approche produit pas tout à fait des arbres car la racine sera remplacée par un circuit de longueur 2 (ce n'est pas gênant si les bornes se recalent toujours en suivant le chemin le plus court).

Le calcul d'une forêt couvrante ne permet pas de résoudre le problème, mais il peut considérablement réduire le nombre d'états du système pour faciliter ensuite une approche du type recuit simulé.

Algorithme de composantes connexes

Une fois le réseau synchronisé, chaque composante connexe aura une heure différente, d'où l'idée d'exécuter un algorithme parallèle de composantes connexes dans lequel l'heure d'une borne représente son numéro de composante connexe provisoire ou son numéro d'étoile par analogie avec l'algorithme 1.1. L'idée est de

reprendre l'algorithme 1.1 et de l'adapter au problème des bornes. On ne peut pas l'utiliser tel quel car on ne peut pas faire l'opération de «*pointer jumping* » sur les pointeurs *Père*, ce qui permettait de trouver et d'accrocher entre elles les étoiles. En effet, le décalage entre une borne et son « grand-père » n'est pas forcément mesuré, et pour l'obtenir, cela nécessiterait d'additionner les deux décalages, ce qui résulterait en de nombreux messages entre les bornes avec surtout le problème de l'accumulation des erreurs.

L'algorithme que nous allons voir est basé sur deux idées : une arête « essaye de synchroniser » ses deux extrémités et si une arête n'arrive pas à synchroniser une de ses deux extrémités sur l'autre, c'est qu'une autre arête essaye de synchroniser cette extrémité dans l'autre sens (c'est l'analogue de l'écriture concurrente dans l'algorithme 1.1). L'algorithme est le suivant : une borne se recale à priori sur toutes ses voisines, si jamais elle s'écarte malgré tout d'une voisine, elle ignore cette voisine le temps de se recaler d'un demi-tour (le chemin maximal qu'elle puisse avoir à faire pour se recaler sur une autre voisine). Pour qu'une borne n'oscille pas entre deux heures (entre deux composantes connexes), elle n'ignore plus jamais par la suite une voisine sur laquelle elle a réussi à se synchroniser et elle lui assigne de plus un poids plus fort qu'aux autres. On dira qu'une voisine est *désactivée* si la borne ignore les décalages avec cette voisine. Une borne ne doit pas non plus ignorer toutes ses voisines, quand il ne lui reste plus qu'une voisine *active*, elle se synchronise sur celle là. Au début, cela ressemble à la synchronisation selon une forêt couvrante : chaque borne se recale sur une seule voisine ; à la fin, une borne se recale sur toutes ses voisines, c'est l'algorithme du gradient. Les détails sont donnés dans l'algorithme 3.6.

w est une constante supérieure à 1 qui module le décalage qu'une borne peut avoir avec sa composante connexe provisoire. Ce décalage ne devrait pas pouvoir dépasser $\frac{T}{2n}$ pour éviter à coup sûr la situation de blocage de la figure 3.4 (n est une borne supérieure à la longueur maximale d'un cycle dans le graphe d'interactions).

Considérons les arêtes qui sortent d'une composante connexe. Si elles tendent à recaler la composante de manières contradictoire, la composante ne se recalera de manière sensible ; cependant, certaines arêtes vont être désactivées petit à petit jusqu'à ce que les arêtes actives restantes ne soient plus contradictoires et la composante va se recaler selon ce qu'elles indiquent de manière sensible cette fois. Le cas le pire est celui où une seule arête sortant de la composante est active : la borne de la composante extrémité de cette arête doit « emmener » toutes les autres, ce qui va prendre un temps proportionnel à $1/\alpha^D$ où D est la longueur maximale d'un plus court chemin reliant cette borne à toute autre borne de la composante. Si cette borne se décale de ε, ses voisines vont se décaler de $\varepsilon\alpha$ au moins, les voisines de ses voisines vont alors se décaler de $\varepsilon\alpha^2$, et ainsi de suite. Les bornes éloignées de la composante connexe vont se recaler $1/\alpha^D$ moins vite (c'est une borne supérieure). Nous avions envisagé $\alpha = 1/2$ en présentant l'algorithme du gradient, une valeur proche de 1 apparaît ici meilleure. Si les bornes se recalent

Algorithme 3.6 Composantes connexes

Données : Les décalages Δ_{ij} pour chaque voisine j de la borne i. Une copie Δ^v_{ij} des décalages. Un ensemble A de voisines actives, un ensemble I de voisines inactives, un ensemble S de voisines synchronisées. A, I, S formera toujours une partition de l'ensemble des voisines de la borne i.

Début

A est initialement l'ensemble de toutes les voisines de la borne i.
$I \leftarrow \emptyset$
$S \leftarrow \emptyset$
$\Delta^v_{ij} \leftarrow \infty$ pour toute voisine j.
Tenir les Δ_{ij} à jour à chaque décalage détecté.
Effectuer à intervalle de temps régulier :
  $G \leftarrow \sum_{j \in A} g(\Delta_{ij}) + \sum_{j \in S} wg(\Delta_{ij})$
  Recaler l'horloge à la vitesse $c_1\, v_r\, G$ d'au plus $c_2\, |G| \,/2$.
  Pour tout $j \in A$ **effectuer**
    Si $\Delta_{ij} \geq \Delta^v_{ij}$ **Alors**
      Mettre j dans I.
    Sinon
      $\Delta^v_{ij} \leftarrow \Delta_{ij}$
    Si $\Delta_{ij} = 0$ **Alors** mettre j dans S.
  Pour tout $j \in I$ **effectuer**
    Si j a été mis dans I depuis un temps supérieur à U, remettre j dans A en posant $\Delta^v_{ij} \leftarrow \infty$.

Fin

toujours à la vitesse maximale v_r et si $U = \frac{T}{2}\frac{1}{v_r\, \alpha^{D'}}$ (où D' est le diamètre du graphe), cet algorithme synchronisera les bornes en temps $O(D'\frac{T}{2}\frac{1}{v_r\, \alpha^{D'}})$ à priori. Ce résultat est purement indicatif. Le raisonnement est théorique car il est remis en cause par l'asynchronisme de l'exécution de l'algorithme sur chaque borne : une arête peut se réactiver à tout moment. Si l'on peut recaler (même de manière approximative) tous les U, l'algorithme devrait synchroniser les bornes à coup sûr. Dans les simulations, $\alpha = 1/2$ convient (le cas le pire est souvent peu probable).

Il est intéressant de considérer le problème de deux bornes oscillant autour de la position où elles sont en opposition de phases dans cet algorithme. Il faut prévoir un test spécial pour désactiver une telle arête si les bornes restent trop longtemps en opposition de phase puisque la comparaison avec le décalage détecté précédemment n'a pas alors le sens attendu. La façon la plus simple de résoudre ce cas particulier consiste certainement à utiliser une fonction g du type de celle de la figure 3.6.

3.3 Simulations

Cette section présente les résultats de quelques simulations de versions simplifiées des algorithmes de recuit simulé 3.5 et de calcul des composantes connexes 3.6.

Présentation du simulateur

Etant donnés les problèmes algorithmiques posés par l'asynchronisme de l'exécution des algorithmes sur chaque borne, il paraît important d'en tenir compte dans des simulations. La modélisation du réseau par une matrice de probabilités permet de « tirer » des communications de décalage au hasard et de simuler l'algorithme de chaque borne de manière assez réaliste : à chaque décalage reçu, une borne doit décider une action en fonction de la valeur récente de ce décalage, de la valeur des décalages avec ses voisines reçus en dernier, du temps écoulé depuis un événement particulier (qui peut être le dernier décalage reçu, une phase de l'algorithme, une décision concernant une borne voisine,...), et du numéro de la voisine dont elle reçoit le décalage.

Il suffit pour cela de considérer que deux communications ne peuvent pas arriver en même temps, les communications sont alors traitées les unes après les autres et sont générées en tirant aléatoirement une borne i de manière uniforme ($1 \leq i \leq n$) et en tirant ensuite une voisine j de i avec la probabilité $P[i, j]$ où P est la matrice de probabilité. Si $j = i$, on considère que la borne ne reçoit pas de communication (cela permet de modéliser le fait que certaines bornes reçoivent moins de décalages que d'autres). Dans les simulations qui suivent, la durée de l'intervalle de temps entre deux communications était tiré aléatoirement de manière uniforme entre 0 et $2M$, où M est proche de la moyenne observée en réalité.

La figure 3.8 montre la fenêtre de l'interface graphique qui permet de visualiser par une horloge l'heure de chaque borne et son évolution au cours de la simulation.

Comparaison des algorithmes

Cette section vise à comparer deux algorithmes, l'un inspiré du recuit simulé, l'autre de l'algorithme de composantes connexes. Nous allons commencer par commenter quelques visualisations en cours d'algorithme sur un graphe régulier : une grille, puis nous donnerons les résultats de nombreuses simulations sur des graphes aléatoires. L'algorithme de recuit simulé utilisé pour les simulations est la variante 3.5 proposée par Daniel Krob. L'algorithme de « composantes connexes » utilisé pour les simulations est une version de l'algorithme 3.6 où le calcul de la moyenne est fait au cours du temps (comme dans l'algorithme 3.1 par rapport à l'algorithme 3.2. La figure 3.9 illustre l'état des bornes d'un réseau en forme de grille à maillage carré en cours de recuit simulé. Les figures 3.10 et 3.11

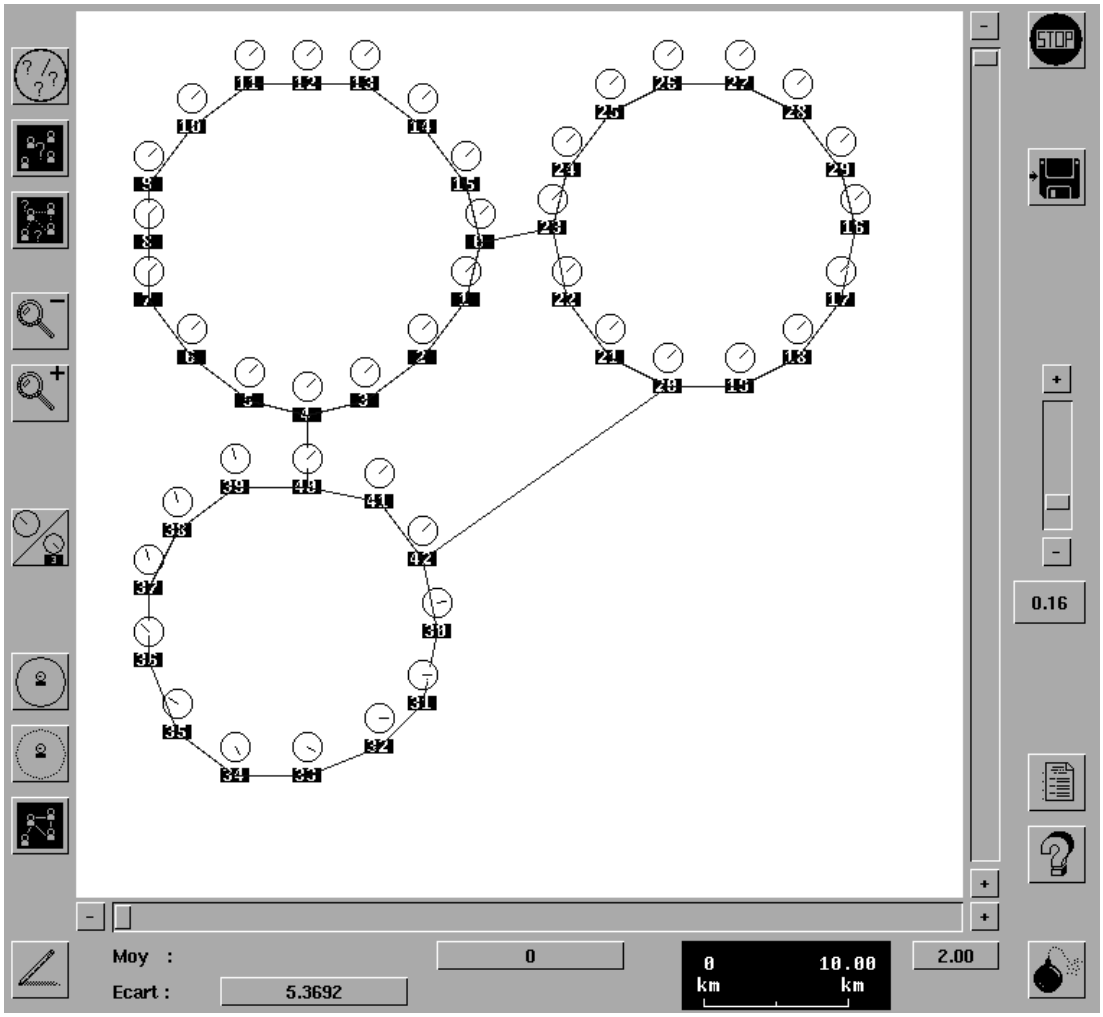


FIG. 3.8 – *Interface du simulateur. Chaque cercle représente l'horloge d'une borne dont l'heure est donnée par l'aiguille à l'intérieur de celui-ci. Les traits entre les étiquettes des horloges illustrent le graphe d'interaction. L'algorithme a ici réussi à synchroniser les bornes des deux anneaux du haut mais pas celles de l'anneau du bas.*

montrent des arrêts sur image de l'exécution de l'algorithme du type composantes connexes sur le même réseau.

La plus grande partie des simulations ont été faites sur des matrices générées aléatoirement. La génération de telles matrices n'est pas évidente car elles doivent être symétriques et la somme des éléments dans chaque ligne doit faire 1 (dans chaque colonne aussi). Une matrice dont la somme des éléments dans chaque ligne est 1 (et dont les éléments sont positifs) s'appelle une matrice MARKOVIENNE. Une manière classique de générer aléatoirement de telles matrices consiste à générer d matrices de permutations $E_1, \ldots, E_d$, c'est-à-dire telles qu'il existe pour

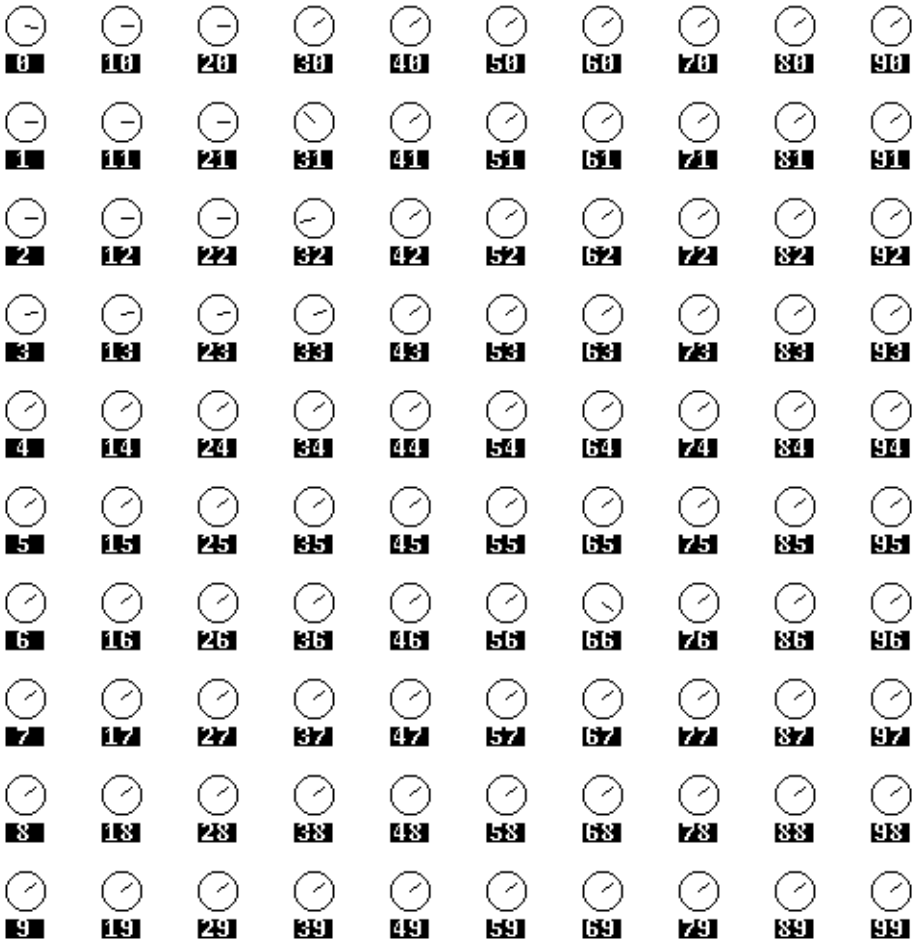


FIG. 3.9 – *L'algorithme du recuit simulé en cours d'exécution. Un « épi » de la borne 32 est en train d'être évacué par le bord du réseau vers le haut. La borne 66 a été mise en opposition de phase par rapport à ses voisines. Elle se recale sur l'heure de ses voisines sans les perturber.*

chaque E_a une permutation π_a telle que $E_a[i,j] = 1$ si et seulement si $\pi_a(i) = j$ (les autres entrées de la matrice sont nulles). On tire ensuite d nombres positifs aléatoires $c_1, \ldots, c_d$. On calcule finalement :

$$P = \frac{1}{\sum_{1 \leq a \leq d} c_a}(c_1 E_1 + \cdots + c_d E_d).$$

Nous avons adapté cette méthode pour obtenir des matrices symétriques où le degré de chaque borne est borné par d. Il suffit pour cela de prendre des

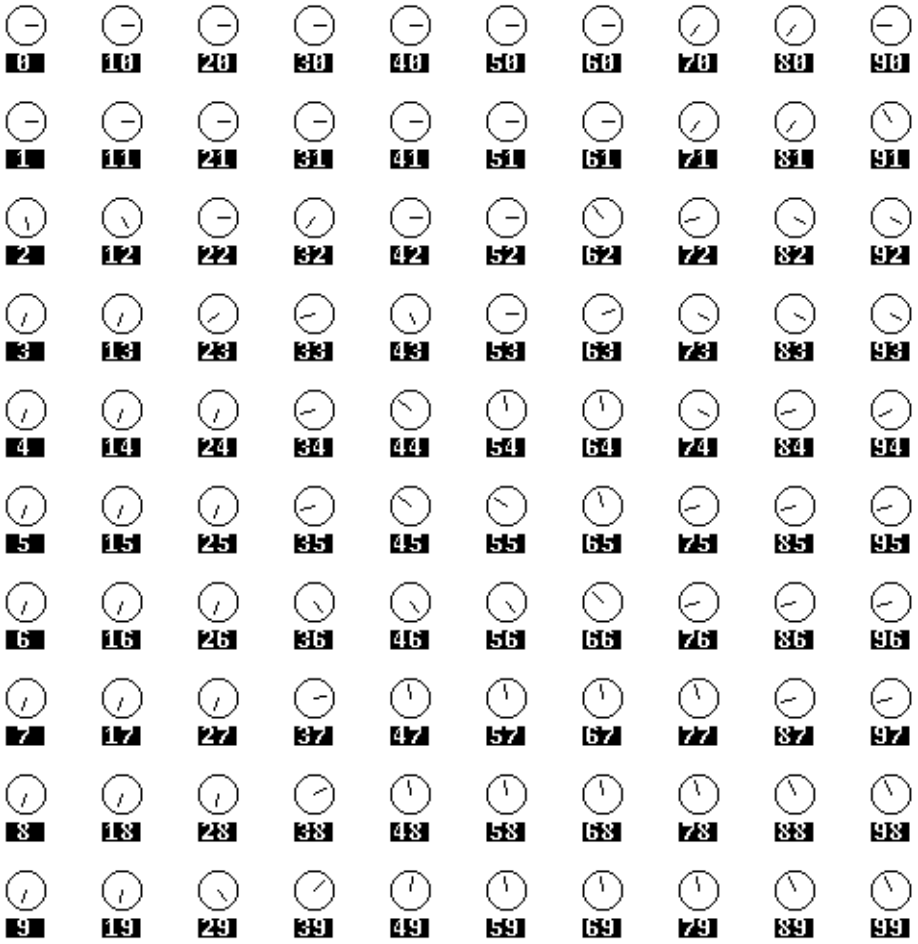


FIG. 3.10 – *L'algorithme du type composantes connexes en cours d'exécution. On peut clairement identifier des sous-ensembles de bornes synchronisées entre elles. Les sous-ensembles s'unissent les uns les autres au cours de l'algorithme. Observer par exemple les influences contradictoires des bornes des sous-ensembles* $\{37, 38, 39\}$ *et* $\{75, 76, 84, 85, 86, 87, 94, 95, 96, 97\}$ *sur celles de* $\{47, 48, 49, 57, 58, 59, 67, 68, 69, 77, 78, 79, 88, 89, 98, 99\}$. *On remarque que de grandes sous-composantes ont tendance à croître en engloutissant des sous-composantes plus petites.*

permutations involutives car

$$\begin{aligned} E_a \text{ est symétrique} &\iff \forall i, j \;\; \pi_a(i) = j \text{ si et seulement si } \pi_a(j) = i \\ &\iff \forall i \pi^2 \text{ est l'identité .} \end{aligned}$$

Les résultats suivants ont été obtenus sur des graphes aléatoires générés avec

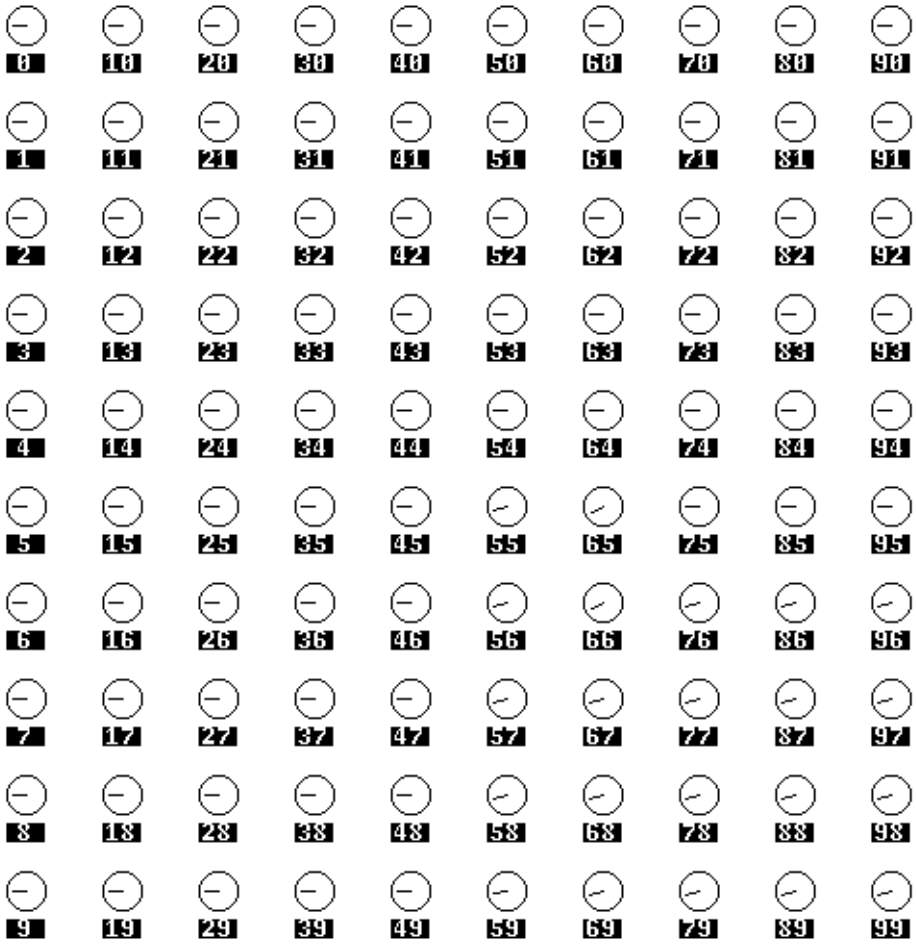


FIG. 3.11 – *Les heures de plusieurs bornes* $(32, 52, 54, 66)$ *ont été modifiée à la main alors que les bornes s'approchaient toutes de la synchronisation sur « 9 heures » par l'algorithme du type composantes connexes. Cela a entraîné la modification de l'heure de tout un ensemble de bornes (le coin en bas à droite). Le phénomène est assez difficile à observer: en général, une borne dont on modifie l'heure se recale assez rapidement sur une de ses voisines sans modifier les heures de ses voisines. Pour pouvoir observer ce phénomène, il a fallu modifier l'heure de plusieurs bornes. Dans certains cas assez rares, la modification d'une borne peut se propager. Cela pourrait être gênant si ce n'était pas aussi improbable que ce que les simulations laissent penser.*

cette méthode. Différents nombres de bornes ont été testés pour différentes valeurs de degré. Chaque point est la moyenne de vingt simulations. L'algorithme "Jeton" est un algorithme de passage de jeton (et qui synchronise selon un arbre couvrant calculé en passant des jetons, une borne se recale sur la première qui lui passe un jeton, elle le fait passer ensuite à ces voisines). Cet algorithme est présenté à

titre comparatif, il donne une idée du temps qu'il faut pour qu'une information traverse tout le réseau. Les figures 3.15, 3.12, 3.13 et 3.14 illustrent ces résultats par des moyennes de temps de convergence des algorithmes en fonction du nombre de sommets des graphes aléatoires utilisés en entrée.

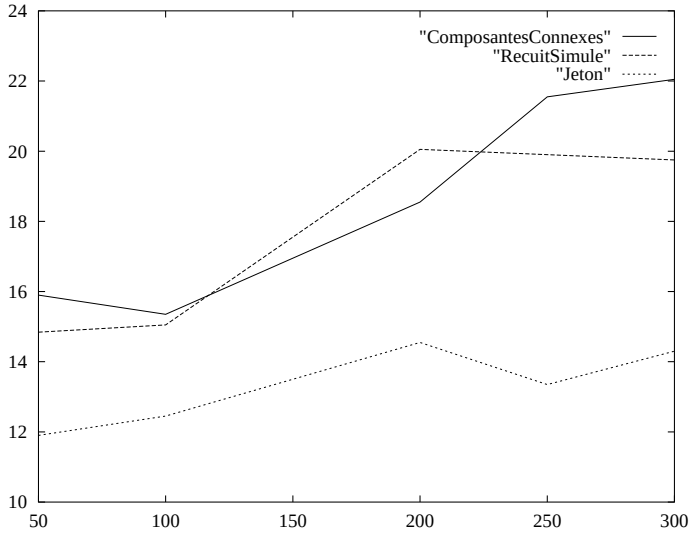


FIG. 3.12 – *Résultats de simulations sur des graphes de degré borné par 5. Différents nombres de bornes (en abscisse) ont été essayés, l'axe des ordonnées donne le temps de synchronisation estimé en heures réelles.*

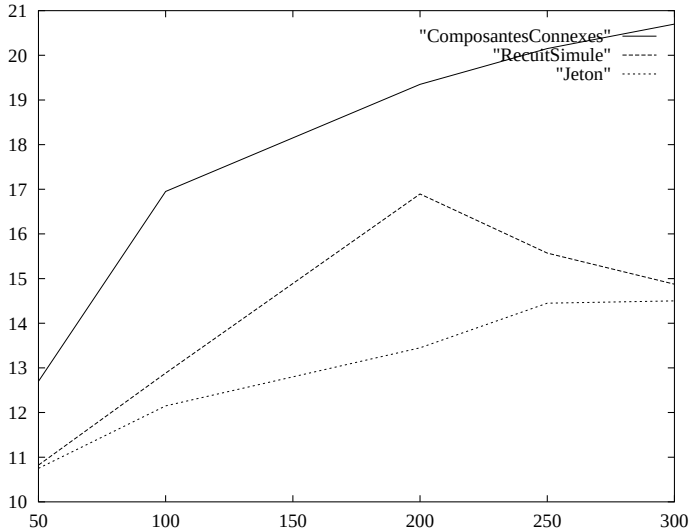


FIG. 3.13 – *Résultats de simulations sur des graphes de degré borné par 10.*

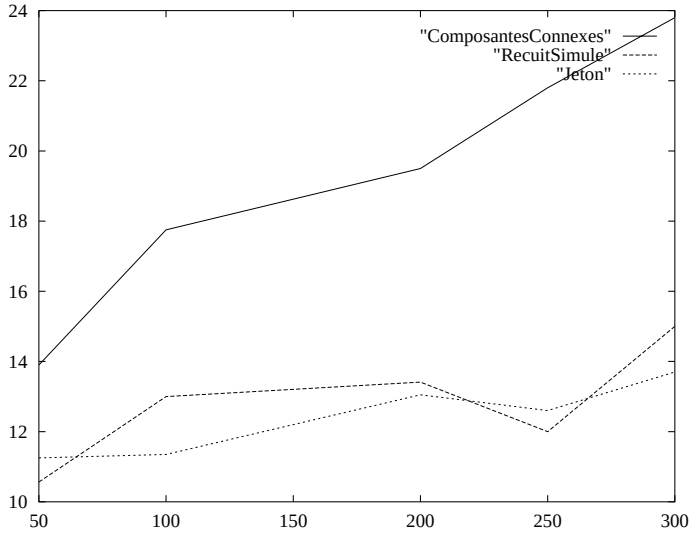


FIG. 3.14 – *Résultats de simulations sur des graphes de degré borné par 15.*

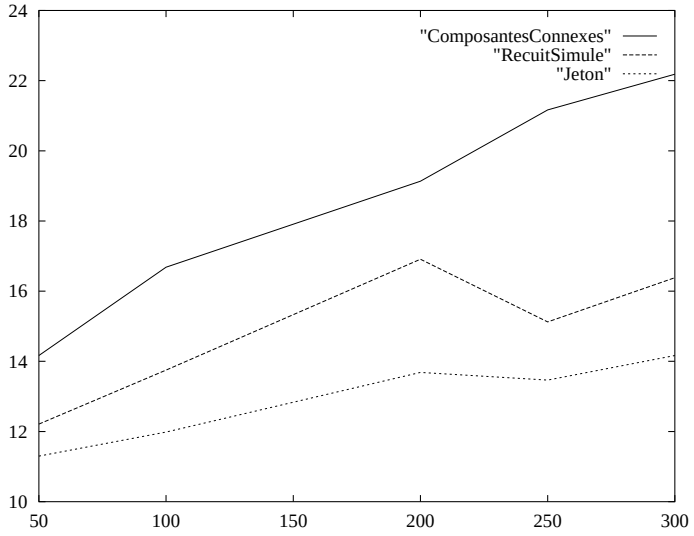


FIG. 3.15 – *La moyenne des résultats précédents.*

Ces résultats montrent que le recuit simulé est très performant en temps sur les graphes de degré important et que l'algorithme de composante connexe est plus rapide sur les graphes de faible degré. Il faut toutefois mettre un bémol à cette observation : sur les 900 simulations effectuées, il y en a huit où le recuit simulé n'a pas réussi à synchroniser les bornes. Cela peut sûrement être résolu en réglant mieux le refroidissement, mais si le graphe entre les bornes n'est pas connu, il est impossible de faire ce réglage. L'algorithme de composantes connexes a synchronisé les bornes dans toutes les simulations. L'algorithme du recuit est

très rapide (aussi rapide que le passage de jeton, voire plus dans les graphes de grand degré) lorsqu'il ne se bloque pas dans un minimum local (parce que sa fonction de refroidissement ne peut pas être réglée pour un graphe précis).

3.4 Quelques problèmes

Les simulations nous confrontent au problème de générer des graphes aléatoires avec certaines propriétés : comment générer des matrices symétriques telles que la somme des éléments de chaque ligne soit 1. Le problème général de la génération aléatoires de structures combinatoires est un problème difficile que l'on sait résoudre quand on sait énumérer (au moins théoriquement) ces structures, c'est par exemple le cas des chemins de MOTZKIN, des animaux dirigés ou des arbres binaires [21]. Peu de classes de graphes usuelles ont été énumérées, on ne connaît par exemple le nombre d'ordres à n sommets que pour n inférieur à douze ou treize.

L'étude du problème de la synchronisation elle-même nous a conduits à un problème de théorie des graphes intéressant : La synchronisation des bornes selon un arbre couvrant permet de résoudre le problème, mais si une borne tombe en panne ou se dérègle, tout le sous-arbre correspondant risque de se désynchroniser du reste des bornes. D'où l'idée d'avoir un arbre couvrant de « secours » : lorsqu'une borne détecte un disfonctionnement de son père, elle se recale sur un autre père dans un deuxième arbre couvrant. Ce deuxième père ne doit pas être un des descendants de la borne car cela créerait un circuit. Si le deuxième père est toujours un ancêtre de la borne, même si plusieurs bornes changent de père, le résultat est toujours un arbre. Ceci n'est pas une condition nécessaire. Comment caractériser de telles paires pour un graphe donné ? Quels sont les graphes qui admettent une telle paire d'arbres couvrants ?

Les algorithmes que nous avons abordés sont loin de fournir des solutions au problème de la synchronisation des bornes d'un réseau GSM. Il reste à faire des études plus poussées sur la stabilité des techniques proposées face au bruitage de la mesure des décalages. De meilleures simulations modéliseraient l'arrivée des communications par un processus de POISSON et bruiteraient les mesures de décalage par un bruit blanc GAUSSIEN. Un algorithme réel combinera sûrement plusieurs idées, il faut de plus trouver comment optimiser les différents paramètres. Les ingénieurs sont sans conteste plus compétents que nous dans ce domaine. D'un point de vue pratique, tout est encore à faire, j'espère que les techniques présentées dans ce chapitre pourront s'avérer utiles.

Conclusion

Nous avons présenté dans ce chapitre quelques techniques algorithmiques pour la résolution de ce problème pratique de synchronisation. Je retiendrai surtout l'idée des composantes connexes qui apporte, je pense, une direction nouvelle dans laquelle chercher des solutions à ce type de problèmes. Les simulations montrent que cette idée peut s'avérer utile : la méthode qu'elle a engendré s'avère comparable en efficacité au recuit simulé. Les techniques de recuit simulé ont été largement étudiées par ailleurs et peuvent sûrement être mieux implantés qu'elles ne l'ont été dans les simulations. Il était important de présenter le recuit simulé dans ce chapitre pour lui comparer l'algorithme issu de l'idée des composantes connexes. Mon but premier n'était pas de résoudre un problème pratique mais de trouver des idées algorithmiques nouvelles à partir d'un problème pratique.

Chapitre 4

Affinage de partition

Ce chapitre a pour but de d'identifier une technique algorithmique utilisée dans quelques algorithmes efficaces de graphes [71, 49, 57]. Cette technique permet de calculer une permutation des sommets d'un graphe qui peut posséder différentes propriétés selon des modifications minimes dans la procédure. Elle consiste à diviser les classes d'une partition de l'ensemble des sommets sans jamais permuter les anciennes classes ou encore à « affiner » itérativement cette partition. Cette technique sera définie dans un cadre plus général encore que la théorie des graphes. Le but de ce chapitre est de mettre sous un même chapeau divers algorithmes efficaces allant de l'orientation transitive d'un graphe à «*quicksort* ». Nous verrons aussi comment cette vision élaborée et unifiée de cette technique permet de généraliser certains algorithmes.

Cette technique me paraît simple à comprendre alors qu'elle est néanmoins très efficace. Elle permet d'obtenir des preuves d'algorithmes assez élégantes et de décrire des algorithmes complexes en quelques lignes. Le paragraphe 4.1 introduit cette technique et montre comment toute une variété d'algorithmes peuvent être décrits assez facilement en termes d'affinage de partition. Quelques généralisations sont même données.

Le paragraphe 4.2 traite de l'orientation transitive et de la reconnaissance des graphes d'intervalles. Tout ce travail découle de la remarque d'une grande similitude entre lex-BFS, un algorithme de parcours de graphe permettant de traiter les graphes chordaux. Les graphes d'intervalles sont des graphes qui permettent de modéliser des événements qui ont une durée dans le temps, ou encore le recouvrement de séquences d'ADN; ils peuvent être représentés par des intervalles d'une droite où deux intervalles sont reliés s'ils se recouvrent. La reconnaissance des graphes d'intervalle consiste à trouver une telle représentation à partir d'une représentation classique du graphe. Dans le cas de l'ADN, cela peut consister à trouver à partir de morceaux d'ADN, dont on peut connaître les recouvrements en comparant les chaînes bases, leur position globale sur le brin d'ADN. Les graphes d'intervalles sont aussi les graphes chordaux dont le complémentaire est transitivement orientable. Leur étude était une application toute désignée par la

découverte de cette ressemblance entre les algorithmes d'orientation transitive et celui de reconnaissance des graphes chordaux.

Le paragraphe 4.3 tente d'aborder l'affinage de partition en parallèle. Il existe un algorithme parallèle de reconnaissance des graphes chordaux qui est basé sur l'affinage de partition [49], mais il est tellement particulier qu'il est difficile d'en généraliser les idées.

4.1 La technique d'affinage de partition

Quelques algorithmes séquentiels récents traitant des graphes de manière efficace (en temps linéaire ou à un facteur logarithmique du linéaire) reposent sur la technique d'affinage de partition introduite ci-après. Il s'agit ici de définir cette technique de manière assez générale pour décrire les algorithmes de recherche de jumeaux [41, 46], de Lex-BFS [71], d'orientation transitive de graphes premiers [57], et des algorithmes qui n'existent pas encore mais que cette technique pourrait aider à concevoir. Une approche similaire du nom de «*graph partitionning* » [73] fait une synthèse des techniques utilisées pour la décomposition modulaire et l'orientation transitive. Encore une autre approche similaire du nom de «*partition refinement* » [81] fait la synthèse d'algorithmes de tri lexicographique et de calcul d'une partition vérifiant certaines propriétés vis à vis d'une relation. Cette section vise à poursuivre ces deux démarches et à étendre comme dans [81] cette idée à des structures plus générales que les graphes : l'affinage de partition peut s'avérer utile dans le traitement de sous-ensembles d'un ensemble en général. Le paragraphe 4.2 présente par exemple l'utilisation de cette technique pour traiter les cliques d'un graphe d'intervalle (ce sont certains sous-ensembles de sommets d'un graphe).

Dans ce chapitre, une *partition* est considérée comme une liste totalement ordonnée de sous-ensembles d'un ensemble E appelés *boîtes*[1], dont l'union est E. *Affiner* une partition consiste à morceler les boîtes en des boîtes plus petites. L'algorithme 4.1 en donne les détails.

La propriété fondamentale de cette procédure élémentaire est qu'aucune boîte de la partition affinée selon S n'intersecte strictement S : toute boîte B'_a de $\mathcal{L}'$ vérifie, soit $B'_a \subseteq S$, soit $B'_a \cap S = \emptyset$. Selon l'utilisation de la procédure, C_a peut être insérée juste à gauche ou juste à droite de B_a à la ligne 1. La figure 4.1 illustre l'affinage d'une partition.

L'affinage selon S prend un temps $O(|S|)$ en utilisant la structure de données suivante. Les éléments de E sont stockés dans une liste doublement chaînée. Chaque boîte forme un intervalle de cette liste et est constituée d'un pointeur vers son premier élément et d'un pointeur vers son dernier élément. Chaque élément contient un pointeur vers sa boîte. Les boîtes sont stockées dans une liste

1. Un algorithme utilisant l'affinage de partition est parfois appelé familièrement un « algorithme de boîtes ».

Algorithme 4.1 Affinage de partition

Données : Une partition $\mathcal{L} = (B_1, \ldots, B_k)$ d'un ensemble E et un sous-ensemble $S \subseteq E$.

Résultat : Une partition $\mathcal{L}' = (B'_1, \ldots, B'_l)$.

Début

  Pour toute *boîte* B_a **effectuer**

    Retirer les éléments de B_a qui sont dans S et les mettre dans une nouvelle boîte C_a.

    Si B_a *est maintenant vide* **Alors**

      Remplacer B_a par C_a.

    Sinon

1   Insérer C_a à côté de B_a.

Fin

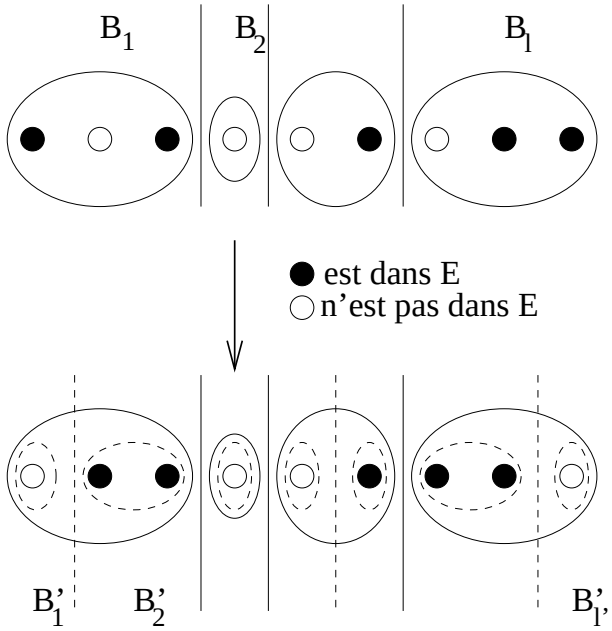


FIG. 4.1 – *Affinage de la partition* $(B_1, \ldots, B_k)$ *en* $(B'_1, \ldots, B'_l)$ *selon le sous-ensemble* S *des éléments en noir.*

doublement chaînée.

Durant l'affinage, chaque élément de S est retiré de la liste et inséré à la fin de la nouvelle boîte correspondant à sa boîte. Cela permet de conserver l'ordre initial des éléments à l'intérieur des boîtes quand S est trié selon cet ordre. On peut aussi maintenir à jour le cardinal des boîtes sans augmenter sensiblement la complexité de la procédure. Quand il n'est pas nécessaire de maintenir l'ordre

sur des éléments à l'intérieur des boîtes, il peut être plus simple de stocker les éléments dans un tableau et d'échanger chaque élément qui doit être retiré et le premier élément (ou le dernier) de sa boîte (les bornes de sa boîte et de la nouvelle boîte correspondante doivent être modifiées en conséquence).

Dans les algorithmes traitant des graphes, E est en général l'ensemble des sommets et S est le voisinage d'un sommet. Un tel sommet est alors appelé un *pivot*. *Pivoter* sur un sommet consiste à affiner une partition selon son voisinage.

Une autre propriété intéressante de cette procédure est que l'affinage de la partition $\mathfrak{L}$ selon le complémentaire de S est presque identique, il suffit d'inverser la règle d'insertion (à droite ou à gauche) de la ligne 1. En ce qui concerne les graphes, cela signifie que l'on peut exécuter la procédure d'affinage sur le graphe complémentaire sans le calculer explicitement, en utilisant seulement les listes d'adjacence du graphe lui-même. Cette propriété est utilisée dans [57] pour la reconnaissance des graphes de permutation qui sont les graphes de comparabilité dont le complémentaire est aussi un graphe de comparabilité.

Plus généralement, un algorithme utilisant cette technique pour travailler sur un graphe peut-être légèrement modifié de manière à travailler sur le graphe complémentaire. Dans [19], une structure de données intermédiaire entre le graphe et son complémentaire est introduite ; elle permet de représenter un graphe en donnant pour chaque sommet soit la liste de ses voisins, soit la liste de ses non voisins. Cette structure de données se prête parfaitement à l'affinage de partition.

Les principaux résultats exposés dans ce chapitre proviennent de la découverte d'une grande similitude entre l'algorithme lex-BFS [71] et l'algorithme d'orientation transitive décrit dans [57].

Permutations lex-BFS

lex-BFS (abréviation de «*lexicographic bread first search* ») est un parcours en largeur de graphe qui visite préférentiellement les sommets possédant des voisins déjà visités. Nous appellerons *permutation lex*-BFS la permutation des sommets produite par une exécution de lex-BFS. lex-BFS a été inventé par Rose, Tarjan et Leuker [71] pour la reconnaissance des graphes chordaux en temps linéaire grâce à la propriété suivante : une permutation lex-BFS est un ordre d'élimination simpliciel des sommets si et seulement si le graphe en entrée est chordal[2]. lex-BFS possède d'autres propriétés liées à l'orientation transitive, qui seront présentées au paragraphe 4.2.

lex-BFS est défini plus formellement de la manière suivante. Les sommets sont numérotés de n jusqu'à 1 au fur et à mesure qu'ils sont visités. Chaque sommet est étiqueté par les numéros de ses voisins déjà visités dans l'ordre décroissant.

2. La définition des graphes chordaux et leur caractérisation par l'existence d'un ordre d'élimination simpliciel seront introduits au paragraphe 4.2. Elles ne sont pas nécessaires à la compréhension de lex-BFS.

Au cours du parcours, le sommet suivant est choisi parmi ceux d'étiquette maximale pour l'ordre lexicographique. Pour des raisons historiques, nous noterons $x_1, \ldots, x_n$ une permutation lex-BFS où x_n est le premier sommet visité et x_1 le dernier sommet visité. L'algorithme 4.2 fournit la version classique de lex-BFS.

Algorithme 4.2 [71] Parcours en largeur « lexicographique » lex-BFS

Données : Un graphe non orienté $G = (V, \mathcal{E})$
Résultat : Un ordre d'élimination simpliciel si G est chordal.
Début
  Pour tout *sommet* $v \in V$ **effectuer** $Etiquette(v) \leftarrow \emptyset$
  Pour $i \leftarrow n$ *jusqu'à* 1 *(par pas de* -1*)* **effectuer**
    Prendre un sommet non visité u d'étiquette maximale.
    $x_i \leftarrow v$
    Pour tout *voisin* v *de* u *non visité* **effectuer**
      $Etiquette(v) \leftarrow Etiquette(v), i$
Fin

Pour prouver la linéarité en temps et en espace de leur algorithme, Rose, Tarjan et Leuker l'implantent avec une technique de type affinage de partition. L'algorithme 4.3 montre comment trouver un sommet d'étiquette maximale en temps constant en maintenant une liste des sommets triés par étiquettes croissantes pour l'ordre lexicographique. Cette liste est en fait une partition de l'ensemble des sommets où les boîtes regroupent les sommets de même étiquette. Cette partition est affinée selon le voisinage du sommet u qui est visité : à l'intérieur d'une boîte, les voisins de u deviennent plus grands que les autres pour l'ordre lexicographique sur les étiquettes. La figure 4.2 donne une illustration de cette phase de l'algorithme et la figure 4.3 montre les différents affinages de partition opérés par lex-BFS sur un exemple.

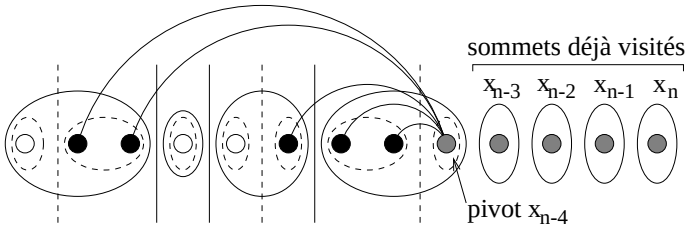


Fig. 4.2 – *Affinage d'une partition en direction d'une permutation lex-*BFS *selon le voisinage du pivot, le sommet en train d'être visité par lex-*BFS.

L'ordre lexicographique vérifie la propriété suivante : si à un moment du parcours lex-BFS, on a $Etiquette(u) < Etiquette(v)$, alors cela est toujours vérifié par la suite. Cette remarque montre que le procédé d'affinage de la partition mène

Algorithme 4.3 Calcul d'une permutation lex-BFS

Données : Un graphe $G = (V, \mathcal{E})$.
Résultat : Une permutation lex-BFS $\mathfrak{L} = x_1, \ldots, x_n$ des sommets.
Début
  $\mathfrak{L} \leftarrow (V)$
  Tant que $\mathfrak{L} = (B_1, \ldots, B_k)$ *contient une boîte non réduite à un singleton* **effectuer**
    Soit B_a la boîte faite de sommets non visités la plus à droite.
    Retirer un sommet u de B_a.
    Insérer $\{u\}$ juste à droite de B_a.
    Pour toute *boîte* B_b *telle que* $b \leq a$ **effectuer**
      Retirer de B_b les sommets voisins de u et les mettre dans une nouvelle boîte C_b.
      Si B_b *est maintenant vide* **Alors**
        Remplacer B_b par C_b.
      Sinon
        Insérer C_b juste à droite de B_b.
Fin

bien à une permutation lex-BFS. Chaque étape d'affinage requiert $O(Degré(v))$ où v est le sommet visité. Remarquons que l'on peut calculer l'étiquette de v au moment où il est visité : c'est l'ensemble des voisins de v apparaissant à sa droite dans la partition. Comme chaque sommet n'est visité qu'une seule fois, la complexité finale est $O(n + m)$.

En inversant la règle d'insertion des boîtes, on peut exécuter lex-BFS sur le complémentaire, et en calculer un ordre d'élimination simpliciel s'il est chordal. Cette remarque pourtant simple n'apparaît pas dans la littérature. Il est même possible de vérifier si le complémentaire d'un graphe est chordal et d'en calculer un arbre de clique si c'est le cas, obtenant ainsi un algorithme linéaire de reconnaissance des graphes dont le complémentaire est chordal (et donc aussi un algorithme de reconnaissance des split graphs qui sont les graphes chordaux de complémentaire chordal).

Abordons maintenant l'algorithme d'orientation transitive présentant quelques similitudes avec lex-BFS.

Orientation transitive

L'algorithme de Mcconnel et Spinrad [57] prend un graphe premier en entrée. Grâce à l'utilisation d'un algorithme de décomposition modulaire linéaire assez lourd, ils peuvent étendre leur algorithme pour calculer une orientation

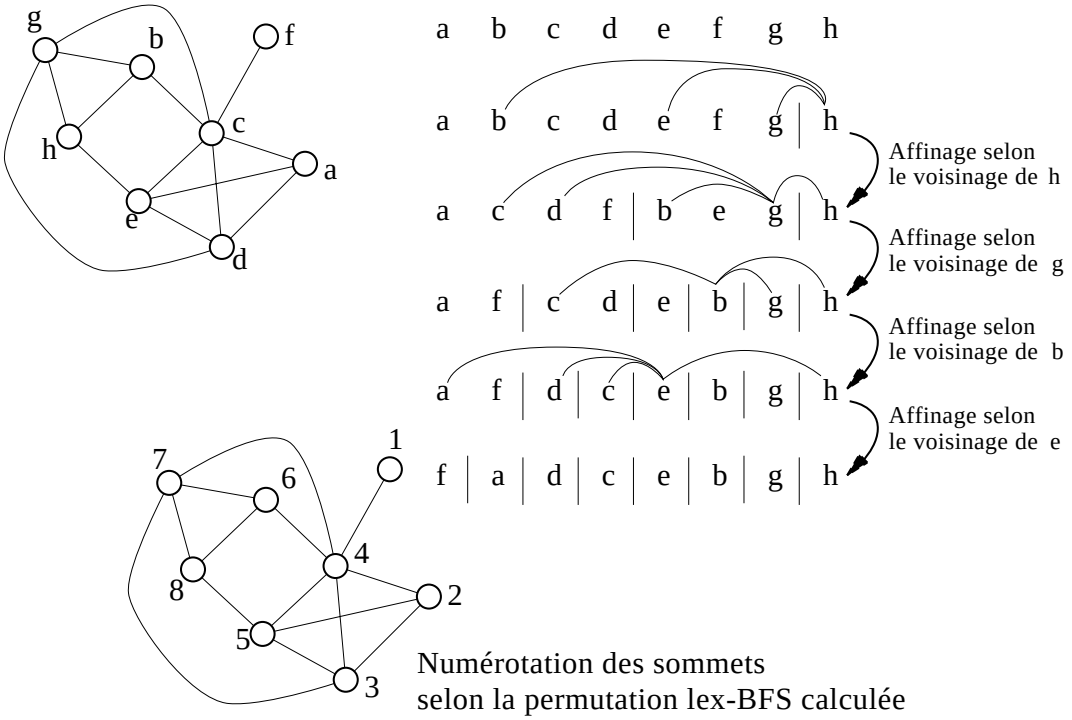


FIG. 4.3 – *Un graphe et l'affinage de partition opéré par une exécution de lex-*BFS. *fadcebgh est la permutation lex-*BFS *calculée.*

transitive de n'importe quel graphe de comparabilité[3]. Cet algorithme calcule une orientation transitive s'il en existe une. Si le graphe en entrée n'est pas un graphe de comparabilité, l'orientation calculée n'est pas transitive. Pour rendre la similitude de cet algorithme avec lex-BFS plus claire, il sera exposé sur le complémentaire, c'est-à-dire comme un algorithme d'orientation transitive du complémentaire du graphe en entrée[4]. L'orientation du complémentaire est implicitement donnée par une extension linéaire de l'ordre résultant de cette orientation transitive, c'est-à-dire une permutation des sommets telle que l'origine d'un arc de l'orientation du complémentaire apparaîsse toujours à gauche de sa destination. Réciproquement, une permutation des sommets induit une orientation du complémentaire en orientant les *non arêtes* (les arêtes du graphe complémentaire) du sommet le plus à gauche dans la permutation vers le sommet le plus à droite. L'algorithme calcule une permutation des sommets qui est une extension linéaire du complémentaire si le graphe en entrée est un graphe premier de *cocomparabilité*

3. Les graphes de comparabilité et le problème de l'orientation transitive sont introduits au paragraphe 1.4.

4. Rappelons que les techniques d'affinage de partition permettent de traiter tout aussi facilement un graphe ou son complémentaire. Les auteurs utilisent eux-mêmes ce fait pour reconnaître les graphes de permutation [57].

(dont le complémentaire est un graphe de comparabilité).

L'algorithme part d'une partition $(V - \{p\}, \{p\})$ où p est un puits possible. Il morcelle répétitivement chaque boîte selon le voisinage d'un sommet u qui n'est pas dans la boîte. Pour obtenir une complexité en $O(n + m \log n)$, u ne sert de pivot que si la taille de sa boîte est au moins deux fois plus petite que celle de la boîte qui le contenait la dernière fois qu'il a servi de pivot. Cela assure que la liste d'adjacence de u est parcourue au plus $\log n$ fois. Quand une boîte vérifie ce critère, l'algorithme va pivoter sur les sommets de cette boîte à la suite. Une liste des boîtes éligibles est maintenue durant le calcul pour éviter de les chercher. Cette stratégie de choix des pivots assure que la boîte la plus grande est un module lorsqu'il n'y a plus de boîtes éligibles. Dans le cas où le graphe en entrée est premier, ce module doit être réduit à un singleton, ce qui signifie que toutes les boîtes sont finalement des singletons. L'algorithme 4.4 donne les détails.

Algorithme 4.4 [**57**] Orientation transitive

Données : Un graphe premier $G = (V, \mathcal{E})$.

Résultat : Une permutation des sommets induisant une orientation transitive de $\overline{G}$ si G est un graphe de cocomparabilité.

Début

- Trouver un puits possible $p \in V$.
- $\mathcal{L} \leftarrow \{V - \{p\}, \{p\}\}$
- *DernièreUtilisation*$(V - \{p\}) \leftarrow \infty$
- *DernièreUtilisation*$(\{p\}) \leftarrow \infty$
- **Tant que** $\mathcal{L} = (B_1, \ldots, B_k)$ *contient une boîte* B_a *telle que* $|B_a| \leq \frac{1}{2}$ *DernièreUtilisation*(B_a) **effectuer**
  - **Pour tout** *sommet* $u \in B_a$ **effectuer**
    - **Pour toute** *boîte* B_b *telle que* $b \neq a$ **effectuer**
      - Retirer de B_b les sommets voisins de u et les mettre dans une nouvelle boîte C_b.
      - **Si** B_b *est maintenant vide* **Alors**
        - Remplacer B_b par C_b.
      - **Sinon**
        - **Si** $b < a$ **Alors**
          - Insérer C_b juste à gauche de B_b.
        - **Sinon**
          - Insérer C_b juste à droite de B_b.
      - *DernièreUtilisation*$(C_b) \leftarrow$ *DernièreUtilisation*(B_b)
  - *DernièreUtilisation*$(B_a) \leftarrow |B_a|$

Fin

La règle d'insertion fixant l'ordre des boîtes de la partition assure que les non

arêtes sont orientées de la gauche vers la droite en partant du fait que le sommet p est un puits possible. La figure 4.4 illustre les relations de forçage direct qui permettent d'affirmer ce fait (elles ne sont pas calculées).

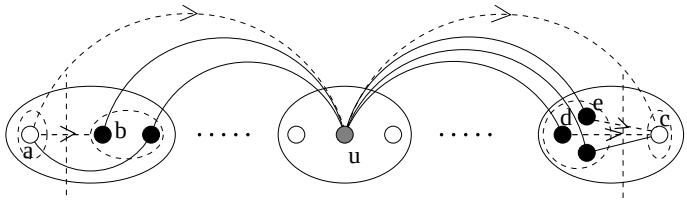


FIG. 4.4 – *Les nouvelles non arêtes inter-boîtes ab, dc et ec sont forcées par au et uc. (u est le pivot ici.)*

Pour trouver une source, MCCONNEL et SPINRAD lancent ce même algorithme en partant d'un sommet quelconque et utilisent un argument similaire sur les relations de forçages pour montrer que le sommet le plus à gauche est un puits possible (l'argument qui dit que la plus grande boîte est un module tient toujours). Nous verrons dans le paragraphe 4.2 comment utiliser lex-BFS pour trouver de manière plus efficace un puits.

Voici quelques arguments formels de la preuve de l'algorithme étaillant l'idée de la figure 4.4. Soit B la plus grande boîte lorsque chaque boîte C de la partition finale vérifie $|C| > \mathit{DernièreUtilisation}(C)$. Tout sommet $u \in V - B$ est dans une boîte C plus petite que B. C est donc au moins deux fois plus petite que la boîte qui contenait à la fois B et C à un moment de l'algorithme. Le pivotage sur u n'a pas morcelé B ; u est donc soit voisin de tous les sommets de B, soit voisin d'aucun. Cela signifie que B est un module.

Si G est un graphe de cocomparabilité, il existe une orientation transitive du complémentaire telle que les non arêtes sont toujours orientées de la gauche vers la droite : toute non arête $\widehat{uv}$ entre deux boîtes distinctes $B_a \ni u$ et $B_b \ni v$ telles que $a < b$ est orientée par uv. La partition $\{V - \{p\}, \{p\}\}$ vérifie bien sûr cette propriété lorsque p est un puits possible. Considérons maintenant une non arête nouvellement inter-boîtes $\widehat{vw}$ résultant du morcelage d'une boîte $B_b \ni v, w$ par un pivot $u \in B_a$. Supposons sans perte de généralité que u est voisin de v mais pas de w. Si $a < b$, alors uw force directement vw, et si $b < a$, alors wu force directement wv. Dans les deux cas, la nouvelle boîte $C_b \ni v$ est insérée de sorte que la non arête soit orientée de la gauche vers la droite. Cela prouve que l'algorithme calcule bien une extension linéaire du graphe complémentaire. Dans le cas où l'algorithme est lancé à partir d'un sommet quelconque, cet argument ne tient que pour les non arêtes issues du sommet p le plus à gauche à la fin de l'algorithme. Elles se forcent les unes les autres et p est donc un puits possible (voir [57] pour plus de détails).

Il existe aussi un algorithme linéaire d'orientation transitive [58] (issu à priori

de l'algorithme de décomposition modulaire [57]). L'article n'étant pas encore publié, je n'ai pas pu le lire[5].

Considérons maintenant le problème de calcul des sommets jumeaux d'un graphe dont la solution linéaire est connue des algorithmiciens mais n'a jamais été publiée en tant que telle.

Calcul des jumeaux

Nous allons maintenant aborder une routine élémentaire en algorithmique des graphes utilisée par exemple dans [41, 46]. Deux sommets d'un graphe sont *jumeaux* s'ils ont même voisinage. Les jumeaux constituent parfois une redondance gênante pour certains algorithmes particuliers [46]. Identifier les jumeaux permet d'épurer un graphe de sorte que deux sommets n'aient pas même voisinage. On distingue deux variantes selon que des jumeaux doivent être voisins ou ne doivent pas l'être. Dans une variante, deux sommets sont jumeaux quand ils ont même voisinage *fermé*, dans l'autre variante, quand ils ont même voisinage *ouvert*. Le voisinage fermé d'un sommet inclue le sommet lui-même, son voisinage ouvert ne le contient pas. Ces deux définitions supposent que le graphe est sans *boucles*, c'est-à-dire sans arête reliant un sommet à lui-même. On peut généraliser ces deux définitions en autorisant les boucles : deux sommets sont jumeaux quand ils ont même voisinage ; dans une variante on ajoutera préalablement chaque sommet à son propre voisinage, dans l'autre, on le retirera. L'algorithme 4.5 donne les détails de cette application très simple de la technique d'affinage de partition.

Algorithme 4.5 [41, 46] Calcul des jumeaux

Données : Un graphe $G = (V, \mathcal{E})$.

Résultat : Une partition $\mathcal{L} = (B_1, \ldots, B_l)$ où deux sommets sont jumeaux si et seulement s'ils sont dans la même boîte.

Début
  $\mathcal{L} \leftarrow \{V\}$
  Pour tout *sommet* $u \in V$ **effectuer**
    Pour toute *boîte* B_a **effectuer**
      Retirer les sommets de B_a qui sont voisins de u et les mettre dans une nouvelle boîte C_a.
      Si B_a *est maintenant vide* **Alors**
        Remplacer B_a par C_a.
      Sinon
        Insérer C_a à côté de B_a.
Fin

5. Je soupçonne qu'il est encore plus compliqué que ceux de décomposition modulaire.

L'ordre des boîtes n'a pas d'importance dans cet algorithme, l'ordre dans lequel on pivote sur les sommets non plus. Si deux sommets u et v ont même voisinage, ils ne seront séparés par aucun pivotage. Réciproquement, si u et v ne sont pas jumeaux, il existe un sommet w relié à l'un mais pas à l'autre ; u et v se retrouveront dans des boîtes différentes après pivotage sur w (w peut être égal à u ou v).

En termes de décomposition modulaire, trouver les jumeaux consiste à trouver les modules qui sont des sous-graphes complets maximaux dans le cas où des jumeaux doivent être voisins, et à trouver les modules qui sont des sous-graphes sans arête maximaux dans l'autre cas. Ces modules correspondent à des nœuds séries dans le premier cas et à des nœuds parallèles dans le second. Il existe une généralisation de la décomposition modulaire aux graphes orientés [24]. Un *module* est alors un ensemble M de sommets d'un graphe orienté $G = (V, \mathcal{E})$ tel que tout sommet u extérieur à M est soit relié à aucun sommet de M, soit relié par un arc sortant uv à tout sommet $v \in M$, soit relié par un arc entrant vu à tout sommet $v \in M$, soit relié à la fois par un arc sortant uv et par un arc entrant vu à tout sommet $v \in M$. Il apparaît alors un nouveau type de nœuds dans la décomposition (le reste est assez similaire) : les nœuds transitifs qui ressemblent à des ordres totaux, les fils d'un tel nœud sont totalement ordonnés et deux sommets de deux fils distincts sont reliés par un arc orienté du sommet du fils le plus petit vers le sommet du fils le plus grand.

On peut généraliser l'algorithme 4.5 pour calculer les modules maximaux d'un graphe orienté qui induisent des ordres totaux, et qui correspondent donc à des nœuds transitifs. Ces modules seront appelés des modules *transitifs* Pour tout sommet u d'un graphe orienté $G = (V, \mathcal{E})$, S_u désignera l'ensemble des sommets v tels que $uv \in \mathcal{E}$ et $vu \notin \mathcal{E}$, I_u désignera l'ensemble des sommets v tels que $vu \in \mathcal{E}$ et $uv \notin \mathcal{E}$, et R_u désignera l'ensemble des sommets v tels que $uv \in \mathcal{E}$ et $vu \in \mathcal{E}$. (Ces ensembles peuvent être calculés à partir des listes d'adjacence par des fusions de listes.) L'algorithme 4.6 permet de calculer une permutation des sommets dont il est facile de déduire les modules transitifs.

L'ordre dans lequel on pivote sur les sommets et l'ordre dans lequel sont maintenues les boîtes sont indifférents. Considérons un module transitif M. Les sommets de M ne peuvent pas être séparés en pivotant sur l'un d'eux puisque pour $u, v \in M$, $uv \in \mathcal{E}$ ou bien $vu \in \mathcal{E}$. Ils ne peuvent pas non plus être séparés en pivotant sur un sommet extérieur à M puisque M est un module. M est donc inclus dans une des boîtes finales. Comme l'ordre des sommets est préservé à l'intérieur des boîtes et comme on a pivoté sur tous les sommets de M, les sommets de M sont triés selon l'ordre total induit par les arcs d'extrémités dans M (c'est un tri « à la quicksort », voir le paragraphe sur quicksort un peu plus loin). Soit u un sommet qui n'est pas dans M. Soit u n'est relié à aucun des sommets de M auquel cas u n'est pas dans la même boîte finale que M. Soit u est relié à tout sommet $v \in M$ par les deux arcs uv et vu auquel cas u n'est pas

Algorithme 4.6 Généralisation du calcul des jumeaux dans un graphe orienté

Données : Un graphe orienté $G = (V, \mathcal{E})$.
Résultat : Une permutation des sommets telle que les sommets de chaque module transitif sont consécutifs et les arcs internes au module ont leur origine à gauche de leur destination.

Début
  $\mathcal{L} \leftarrow \{V\}$
  Pour tout *sommet* $u \in V$ *de boîte* B **effectuer**
    Affiner la partition selon S_u, selon I_u, et selon R_u en préservant l'ordre des sommets (cela requiert de trier S_u, I_u et R_u).
    B est coupée de manière spéciale : les sommets appartenant à $B \cap I_u$, $\{u\}$, et $B \cap S_u$ sont mis dans la même boîte selon cet ordre (les sommets dans $B \cap I_u$ à gauche, les sommets dans $B \cap S_u$ à droite, et u entre les deux).
Fin

dans la même boîte finale que M. Soit u est relié à tout sommet $v \in M$ par uv mais pas par vu auquel cas u apparaît avant les sommets de M si jamais il est dans la même boîte finale. Soit u est relié à tout sommet $v \in M$ par vu mais pas par uv auquel cas u apparaît après les sommets de M si jamais il est dans la même boîte finale. Dans tous les cas, aucun sommet extérieur à M ne vient s'intercaler entre les sommets de M.

Chaque sommet est utilisé une seule fois comme pivot, mais comme il faut trier ses listes d'adjacence (au moment où il sert de pivot), l'algorithme requiert à priori un temps $O(n + m \log n)$ (ou $O(n^2)$ si on fait des tris du géomètre). (Les sommets sont numérotés au moment de la création d'une boîte du type $B \cap I_u$, $\{u\}$, $B \cap S_u$, de la gauche vers la droite.) On ne peut pas trier une fois pour toutes les listes d'adjacence au début de l'algorithme car l'ordre des sommets dans la boîte du pivot change (à chaque affinage).

Deux sommets consécutifs u et v (u étant juste à gauche de v) dans la permutation finale sont dans un même module transitif si et seulement si $S_u - \{u\} = S_v \cup \{v\}$ et $I_u \cup \{u\} = I_v - \{v\}$ et $R_u = R_v$. Cela permet de calculer les modules transitifs à partir de la permutation calculée par l'algorithme 4.6 en temps linéaire.

Nous allons maintenant voir que les parcours en largeur classiques peuvent aussi être implantés par affinages de partition, ce n'est pas le seul cas de lex-BFS.

Parcours en largeur

En ne coupant que la boîte la plus à gauche dans l'algorithme 4.3, on obtient un parcours en largeur classique où les boîtes regroupent des sommets qui sont à

même *distance* du sommet de départ du parcours (la distance entre deux sommets est la longueur du plus court chemin les reliant). Cela donne l'algorithme 4.7

Algorithme 4.7 Parcours en largeur et calcul des distances à un sommet

Données : Un graphe $G = (V, \mathcal{E})$ et un sommet v_0.

Résultat : La distance $d(u, v_0)$ pour chaque sommet $u \in V$.

Début

$\mathcal{L} \leftarrow (V - \{v_0\}, \{v_0\})$
Distance$(\{v_0\}) \leftarrow 0$
Distance$(V - \{v_0\}) \leftarrow \infty$
BoîteCourante $\leftarrow \{v_0\}$
Tant que *BoîteCourante n'est pas la boîte la plus à gauche* **effectuer**
  Pour tout *sommet* $u \in$ *BoîteCourante* **effectuer**
    Retirer de B_1, la boîte la plus à gauche de la partition, les sommets voisins de u et les mettre dans une nouvelle boîte C.
    Insérer C juste à droite de B_1.
    Distance$(C) \leftarrow$ *Distance*(*BoîteCourante*) $+ 1$
  BoîteCourante devient la boîte juste à gauche de *BoîteCourante*.
Pour tout *sommet* u *de boîte* B **effectuer** $d(u, v_0) \leftarrow$ *Distance*(B)

Fin

Si les pivots sont pris de la droite vers la gauche dans *BoîteCourante*, la permutation finale (lue de la droite vers la gauche) donne l'ordre de parcours des sommets selon le parcours en largeur correspondant. Si on veut seulement que les boîtes reflètent les classes de sommets à même distance de v_0, il suffit d'allonger les étapes d'affinage de la partition : le morcelage de B_1 est terminé une fois que les listes d'adjacence de tous les sommets de *BoîteCourante* ont été parcourues, ce qui consiste à affiner selon l'union des voisinages des sommets de *BoîteCourante*.

Remarquons qu'un parcours en profondeur ne peut pas être interprêté par des affinages de partition car il faudrait changer l'ordre des boîtes (les voisins du deuxième sommet visité doivent être visités avant les autres voisins du premier sommet visité, même s'ils ne sont pas voisins du premier sommet visité).

Examinons enfin quelques propriétés générales de l'affinage de partition.

Considération générale sur l'affinage de partition

Considérons un ensemble E et supposons que l'on affine la partition (E) consécutivement avec des sous-ensembles $S_1, \ldots, S_j$ en insérant toujours une nouvelle boîte juste à droite de celle dont elle est issue. L'ordre dans lequel les S_i sont utilisés n'influe pas sur les propriétés que nous allons dégager ici. Soit $(B_1, \ldots, B_k)$ la partition finale. Chaque élément $u \in E$ est associé à un

sous-ensemble $C_u \subseteq \{S_1, \ldots, S_j\}$: l'ensemble des S_i qui le contiennent[6] (voir la figure 4.5). La partition finale jouit des propriétés suivantes :

1. Deux éléments u et v sont dans la même boîte finale si et seulement si $C_u = C_v$.
2. En ordonnant les éléments de manière quelconque à l'intérieur de chaque boîte, on obtient une permutation $x_1, \ldots, x_n$ telle que $C_{x_1}, \ldots, C_{x_n}$ est une extension linéaire de $(\{C_u \mid u \in V\}, \subset)$.

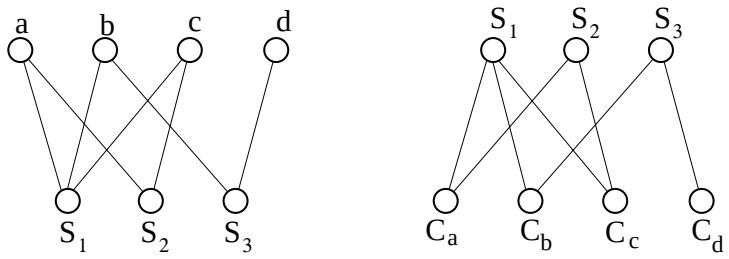


FIG. 4.5 – *(i) Un graphe biparti permet de représenter les parties d'un ensemble. On a par exemple ici $S_2 = \{a, c\}$ et $C_b = \{S_1, S_3\}$. (ii) En retournant cette représentation, on lit l'ensemble des parties qui contiennent chaque sommet.*

En effet, si deux éléments u et v vérifient $C_u \neq C_v$, alors il existe un sous-ensemble S_i qui contient un sommet et pas l'autre ; l'affinage selon cet ensemble placera u et v dans des boîtes différentes si cela n'avait pas encore été fait. Si $C_u = C_v$, alors aucun affinage ne séparera u et v.

Si deux éléments u et v vérifient $C_u \subset C_v$, alors u et v resteront dans la même boîte jusqu'à l'affinage selon un sous-ensemble S_i contenant v mais pas u et v sera alors placé à droite de u ; cela ne peut pas changer par la suite.

La propriété 1 est utilisée dans la reconnaissance des jumeaux qui sont les sommets de même voisinage (l'algorithme consiste à affiner selon les voisinages). La propriété 2 fournit un algorithme permettant de calculer une extension linéaire d'un ensemble $\mathcal{A}$ de parties d'un ensemble E en affinant la partition $(\mathcal{A})$ selon les sous-ensembles de parties $D_u = \{S \in \mathcal{A} \mid u \in S\}$, $u \in E$ (il suffit de remarquer que $C_S = \{D_u \mid S \in D_u\}$ est isomorphe à S).

On peut associer un « arbre de pivotage » à toute exécution d'une procédure d'affinages de partitions où chaque nœud de l'arbre est une boîte apparue au cours de l'algorithme et où ses fils sont les sous-boîtes issues de son morcelage. Les arêtes de l'arbre sont étiquetées par le pivot qui a morcelé la boîte père. Cet arbre peut permettre d'analyser les algorithmes d'affinage de partition de manière plus fine, mais l'intérêt de la technique est d'oublier cet arbre quand seul l'ordre final de ses feuilles compte, ce qui rend l'algorithme plus simple.

6. Cette construction s'appelle le dual en théorie des hypergraphes.

4.2 lex-BFS et orientation transitive

Cette section est consacrée à la découverte d'un lien profond entre lex-BFS et l'orientation transitive. Cela nous permettra d'en déduire un algorithme très simple et efficace d'orientation transitive et un algorithme linéaire de reconnaissance des graphes d'intervalles. Ces deux algorithmes sont tous les deux constitués d'une lex-BFS suivie d'un algorithme de type affinage de partition. Avant d'entrer dans le vif du sujet, il nous faut tout d'abord introduire quelques définitions concernant les graphes chordaux.

Graphes chordaux

Un graphe non orienté est dit *chordal* (ou *triangulé*) si et seulement si tout cycle de longueur supérieur ou égal à 4 possède une *corde*, c'est-à-dire une arête joignant deux sommets non consécutifs sur le cycle. Les caractérisations suivantes des graphes chordaux donnent une idée des nombreuses propriétés algorithmiques dont ils font preuve.

Les graphes chordaux sont caractérisés par l'existence d'un *ordre d'élimination simpliciel* de ces sommets. Une *clique* est un ensemble des sommets induisant un sous-graphe complet. $x_1, \ldots, x_n$ est un ordre d'élimination simpliciel d'un graphe $G = (V, \mathcal{E})$ si et seulement si le voisinage de chaque sommet x_i est une clique du sous-graphe induit $G_{\{x_i,\ldots,x_n\}}$. L'algorithme de reconnaissance de Rose, Tarjan et Leuker repose sur cette caractérisation. Ils ont montré qu'une permutation lex-BFS d'un graphe chordal est aussi un ordre d'élimination simpliciel. Les cliques maximales (pour l'inclusion) se calculent facilement à partir d'un tel ordre. La figure 4.6 donne un exemple.

Cela nous conduit à la seconde caractérisation [35] : un graphe est chordal si et seulement s'il existe un arrangement de ses cliques maximales en arbre de sorte que les cliques maximales contenant un sommet donné induisent toujours un sous-arbre connexe. Un tel arbre s'appelle un *arbre de cliques*. Les graphes d'intervalles sont les graphes chordaux admettant un arbre de clique qui est une chaîne ou de manière équivalente une permutation de ses cliques maximales telle que l'ensemble des cliques maximales contenant un sommet donné soient consécutives. Une telle chaîne s'appelle une *chaîne de cliques*. Un intervalle de la chaîne est donc associé à chaque sommet : l'intervalle des cliques le contenant. Il en résulte une *représentation intervallaire* : deux sommets sont reliés si et seulement s'il existe une clique maximale les contenant tous les deux, c'est-à-dire si et seulement si leurs intervalles associés se recouvrent.

L'algorithme 4.8 du à [33] calcule un arbre de clique durant l'exécution de lex-BFS, en conservant une complexité linéaire en temps et en espace. Il permet aussi de vérifier que le graphe en entrée est chordal. L'algorithme de reconnaissance des graphes d'intervalles qui est présenté plus loin utilise un arbre de cliques.

Voici quelques indications sur l'exactitude de cet algorithme, les détails sont

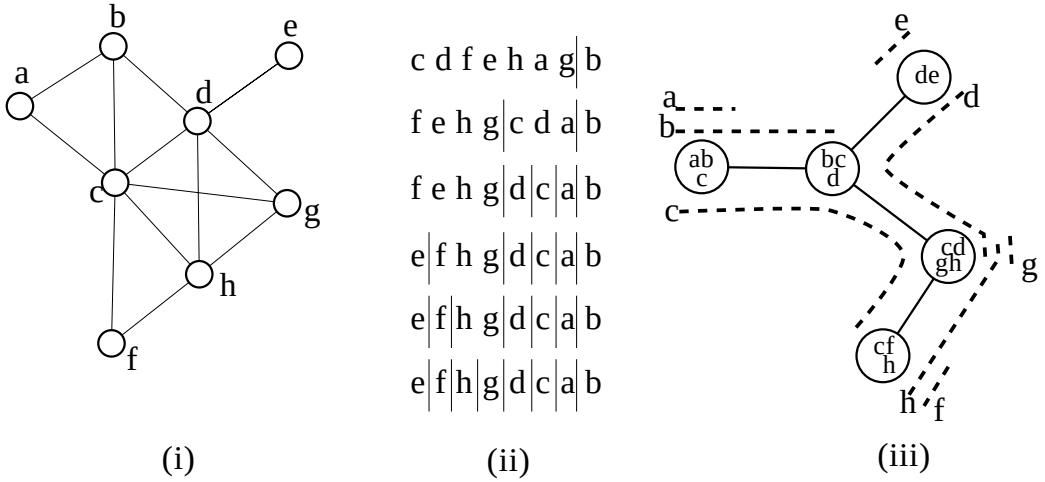


FIG. 4.6 – *(i) Un graphe chordal. (ii) Les affinages de partitions successifs conduisant à un ordre d'élimination simpliciel calculé par lex-*BFS*. L'ensemble des voisins de h sur sa droite est* $\{c, d, g\}$ *qui induit bien une clique. (iii) Un arbre de clique représentant ce graphe. Deux sont reliés si et seulement si les deux sous-arbres associés se recouvrent.*

dans [33]. Puisque lex-BFS calcule un ordre d'élimination simpliciel, quand un sommet u est visité, $\mathit{Etiquette}(u) \cup \{u\}$ est une clique. Si elle n'est pas maximale, tout autre sommet v appartenant à une clique maximale contenant $\mathit{Dernier}(u) \cup \{u\}$ est un voisin de u non encore visité vérifiant $\mathit{Etiquette}(u) \cup \{u\} \subseteq \mathit{Etiquette}(v)$. Remarquons que le dernier sommet visité v d'une clique maximale C vérifie $\mathit{Etiquette}(v) \cup \{v\} = C$. En considérant les étiquettes des sommets (au moment où ils sont visités), les séquences strictement croissantes pour l'inclusion correspondent aux cliques maximales. Cela fournit une méthode classique pour calculer les cliques maximales à partir de n'importe quel ordre d'élimination simpliciel [71]. C'est ce que fait cet algorithme. Le test d'inclusion prend un temps $O(|\mathit{Etiquette}(u)|) = O(\mathit{Degré}(u))$. L'ensemble de ces tests permet aussi de s'assurer que le graphe en entrée est bien chordal.

Montrons maintenant pourquoi T est un arbre de cliques, c'est-à-dire pourquoi les cliques maximales contenant un sommet donné u induisent un sous-arbre connexe. Remarquons que $C(u)$ est la première clique maximale découverte qui contient u et que $\mathit{Dernier}(v)$ est à tout moment le dernier voisin visité de v. Il suffit de montrer que le chemin de T reliant une clique C contenant un sommet donné u à $C(u)$ passe par des cliques contenant toujours u. Considérons le sommet visité v au moment de la création de $C = C(v)$. Si le père de C dans T est $C(u)$, il n'y a rien à prouver. D'un autre côté, si $\mathit{Dernier}(v)$ et u sont deux sommets distincts, ils doivent être reliés puisqu'ils sont tous les deux voisins de v et apparaissent à droite de v dans l'ordre d'élimination simpliciel calculé. Le père $C(\mathit{Dernier}(v))$ de

Algorithme 4.8 [33] lex-BFS et calcul d'un arbre de cliques

Données : Un graphe $G = (V, \mathcal{E})$.

Résultat : Détermine si le graphe est chordal et si c'est le cas calcule un ordre d'élimination simpliciel $x_1, \ldots, x_n$ et un arbre de clique $T = (\mathcal{I}, \mathcal{F})$ où $\mathcal{I}$ est l'ensemble des cliques maximales.

Début

Pour tout *sommet* $u \in V$ **effectuer** $Etiquette(v) \leftarrow \emptyset$
$EtiquettePrecedente \leftarrow \emptyset$
$j \leftarrow 0$
Pour $i = n$ *jusqu'à* 1 **effectuer**
Prendre un sommet u d'étiquette maximale pour l'ordre lexicographique.
Si $EtiquettePrecedente \not\subset Etiquette(u)$ **Alors**
$j \leftarrow j + 1$
Créer la clique maximale $C_j \leftarrow Etiquette(u) \cup \{u\}$.
$C(Dernier(u))$ est le père de C_j dans T.
L'arête $C_j\widehat{C(Dernier(u))}$ de l'arbre est associée au séparateur minimal $S_j = C_j \cap C(Dernier(u)) \leftarrow Etiquette(u)$.
Sinon
$C_j \leftarrow C_j \cup \{u\}$
Pour tout *voisin* v *de* u **effectuer**
$Etiquette(v) \leftarrow Etiquette(v) \cup \{u\}$
$Dernier(v) \leftarrow u$
$EtiquettePrecedente \leftarrow Etiquette(u)$
$x_i \leftarrow u$
$C(u) \leftarrow j$

Fin

C dans T contient donc u. On conclue en itérant ce procédé avec $C(Dernier(v))$ et ainsi de suite jusqu'à ce que $Dernier(Dernier(\cdots Dernier(v)\cdots)) = u$.

Chaque arête $\widehat{C_jC_k}$ de l'arbre est associée à un ensemble $S_j = Etiquette(u)$ où C_j est la clique créée en visitant u. Cet ensemble s'appelle un *séparateur minimal* entre C_j et C_k (parce que c'est un ensemble minimal dont l'effacement déconnecte des sommets des deux cliques). Nous utiliserons uniquement le fait que c'est l'intersection de C_j et C_k. Comme les voisin déjà visités de u forment une clique et comme $Dernier(u)$ est le dernier d'entre-eux, on obtient $Etiquette(u) \subseteq Etiquette(Dernier(u)) \cup \{Dernier(u)\}$. Cela implique $Etiquette(u) \subseteq C_k$ et donc $Etiquette(u) \subseteq C_j \cap C_k$. L'inclusion $C_j \cap C_k \subseteq Etiquette(u)$ découle de la définition de $Dernier(u)$.

L'algorithme de reconnaissance des graphes d'intervalles présenté un peu plus

loin utilise la propriété suivante.

Lemme 38 ([35]) *Dans un arbre de cliques d'un graphe chordal, si deux cliques maximales contiennent toutes les deux un sommet u, alors toutes les arêtes du chemin les reliant sont associées à des séparateurs minimaux contenant tous u.*

Permutations lex-BFS et orientation transitive

Rappelons que les modules d'un graphes sont aussi ceux du complémentaire. Pour relier la notion de module à celle de boîte, notons que si une boîte d'une partition de l'ensemble des sommets d'un graphe contient un module alors ce module est encore entièrement inclus dans une boîte après pivotage sur un sommet extérieur à cette boîte. Ce principe est à la base des algorithmes linéaires de décomposition modulaire [15, 57]. Si l'on examine la boîte la plus à gauche dans lex-BFS, elle est toujours coupée selon des pivots extérieurs à celle-ci jusqu'au moment où un de ses sommets est visité. A ce moment de l'algorithme, la boîte la plus à gauche est module car on a pivoté sur tous les sommets extérieurs à celle-ci. Les deux résultats suivants expriment de manière plus fine le lien entre lex-BFS et l'orientation transitive. Ils sont le point d'orgue des algorithmes présentés ensuite.

Lemme 39 *Si M est un module d'un graphe G, alors toute permutation lex-*BFS *de G induit une permutation lex-*BFS *du sous-graphe G_M induit par M.*

Preuve. Soit M un module de G. Quand un sommet $u \in M$ est visité par lex-BFS, son étiquette $Etiquette(u)$ est supérieure ou égale à l'étiquette $Etiquette(v)$ de tout sommet $v \in M$ non encore visité pour l'ordre lexicographique. Comme M est un module, on a $Etiquette(u) - M = Etiquette(v) - M$, ce qui implique que $Etiquette(u) \cap M$ est plus grand que $Etiquette(v) \cap M$ dans l'ordre lexicographique. Cela signifie que la permutation lex-BFS de G induit une permutation des sommets dans M qui est une permutation lex-BFS de G_M. □

Théorème 40 *Soit M un module d'un graphe de cocomparabilité G et soit $u \in M$ le dernier sommet visité lors d'une exécution de lex-*BFS *sur G. Alors il existe une orientation transitive de $\overline{G_M}$ où u est un puits. Si le graphe est de plus chordal (c'est alors un graphe d'intervalles), alors les sommets appartenant à $Etiquette(u) \cap M$ (c'est-à-dire les voisins de u dans M) sont aussi des puits dans cette orientation.*

Ce théorème découle de la relation de forçage introduite par [34] (voir le paragraphe 1.4). La figure 4.7 illustre certaines relations de forçage apparaissant au cours de l'exécution de lex-BFS.

Rappelons qu'il ne peut pas y avoir de relation de forçage entre un arc interne à un module et un arc sortant du module, ce qui implique que l'orientation obtenue en inversant l'orientation des arêtes internes à un module est encore transitive.

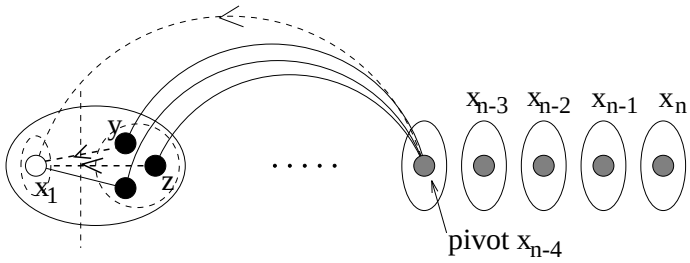


FIG. 4.7 – *Relation de forçage et lex*-BFS. $x_{n-4}x_1$ *force* yx_1 *et* zx_1 *où* x_1 *est le dernier sommet visité.*

Preuve. En utilisant le lemme 39, il suffit de prouver que pour un graphe de cocomparabilité premier, le dernier sommet visité x_1 lors d'une exécution de lex-BFS est un puits ou une source[7] (la permutation lex-BFS calculée sera notée $x_1, \ldots, x_n$). Soit $\mathcal{F}$ une orientation transitive des non arêtes. Remarquons que x_1 est toujours dans la boîte la plus à gauche au cours de l'exécution de lex-BFS. Comme le graphe est premier, x_n n'est pas voisin de tous les autres sommets ($\{x_1, \ldots, x_{n-1}\}$ serait un module dans le cas contraire). $\widehat{x_1x_n}$ est donc une non arête. Supposons que son orientation dans $\mathcal{F}$ est x_nx_1. D'après la relation de forçage, on déduit par récurrence sur les boîtes successivement plus petites qui contiennent x_1 que si B_1' est la boîte contenant x_1 à un moment de l'algorithme, alors toutes les non arêtes entre $V - B_1'$ et x_1 sont orientées vers x_1. Remarquons que le pivot qui a coupé la boîte B_1 contenant précédemment x_1 en B_1' et B_2' n'était pas dans B_1 car cela impliquerait que B_1 est un module puisque tous les sommets n'appartenant pas à B_1 auraient déjà servi de pivot. La dernière boîte contenant x_1 est $\{x_1\}$, x_1 est donc un puits. Si l'orientation de la non arête $\widehat{x_1x_n}$ était en fait x_1x_n, le même raisonnement montre que x_1 est une source, c'est-à-dire un puits de l'orientation transitive inverse $\mathcal{F}^{-1}$.

Considérons maintenant le cas où le graphe est de plus chordal. Grâce au lemme 39, le cas général se déduit à nouveau du cas premier. Supposons donc G chordal, premier et de cocomparabilité. Le dernier sommet visité x_1 est simpliciel : son voisinage est une clique. On peut prouver comme précédemment que x_1 est un puits d'une des deux orientations transitives de $\overline{G}$. Soit u un sommet extérieur à cette clique. L'orientation de toutes les non arêtes reliant un sommet de la clique à u sont forcées par ux_1. Les voisins de x_1 sont donc aussi des puits □

Orientation transitive

Le théorème 40 permet d'améliorer l'algorithme d'orientation transitive 4.4 de MCCONNEL et SPINRAD [57] qui prend un graphe premier en entrée, pour calculer une orientation transitive du complémentaire d'un graphe de cocomparabilité

7. Un graphe premier n'a que deux orientations transitives $\mathcal{F}$ et $\mathcal{F}^{-1}$, les puits de l'une sont les sources de l'autre.

quelconque sans utiliser le lourd outillage de la décomposition modulaire.

Quand l'algorithme est bloqué, la plus grande boîte B est un module. Pour poursuivre l'algorithme, il suffit de couper B en $B - \{u\}, \{u\}$ où u est le dernier sommet visité par une exécution préalable de lex-BFS puisque c'est un puits possible du module d'après le théorème 40. L'algorithme 4.9 regroupe toutes les idées introduites sur l'orientation transitive.

Algorithme 4.9 Orientation transitive

Données : Un graphe premier $G = (V, \mathcal{E})$.

Résultat : Une permutation des sommets induisant une orientation transitive de $\overline{G}$ si G est un graphe de cocomparabilité.

Début

Calculer une permutation lex-BFS $x_1, \ldots, x_n$ des sommets.
$\mathcal{L} \leftarrow (\{x_1, \ldots, x_n\})$
DernièreUtilisation$(\{x_1, \ldots, x_n\}) \leftarrow \infty$
Tant que $\mathcal{L} = (B_1, \ldots, B_k)$ *contient une boîte non réduite à un singleton* **effectuer**
  Si *il existe une boîte* B_a *telle que* $|B_a| \leq \frac{1}{2}$ *DernièreUtilisation*(B_a) **Alors**
    Pour tout *sommet* $u \in B_a$ **effectuer**
      Pour toute *boîte* B_b *telle que* $b \neq a$ **effectuer**
        Retirer de B_b les sommets voisins de u et les mettre dans une nouvelle boîte C_b.
        Si B_b *est maintenant vide* **Alors**
          Remplacer B_b par C_b.
        Sinon
          Si $b < a$ **Alors**
            Insérer C_b juste à gauche de B_b.
          Sinon
            Insérer C_b juste à droite de B_b.
        DernièreUtilisation$(C_b) \leftarrow$ *DernièreUtilisation*(B_b)
      DernièreUtilisation$(B_a) \leftarrow |B_a|$
    Sinon
      Soit B la plus grande boîte et u le sommet de B le plus à gauche dans la permutation lex-BFS.
      Retirer u de B et insérer $\{u\}$ juste à droite de B.
      DernièreUtilisation$(u) \leftarrow \infty$

Fin

Quand toutes les classes éligibles ont été utilisées, la plus grande classe est fa-

cile à trouver, c'est la seule qui n'a pas été éligible. Pour trouver le dernier sommet visité par lex-BFS dans une boîte, il suffit de conserver l'ordre de la permutation lex-BFS à l'intérieur des boîtes, ce sommet est alors le plus à gauche dans la boîte. La complexité de cet algorithme est la même que celle de l'algorithme d'orientation 4.4 pour les mêmes raisons.

Cet algorithme calcule une extension linéaire d'une orientation de $\overline{G}$ quand G est un graphe de cocomparabilité en temps $O(n + m \log n)$. En inversant la règle d'insertion des boîtes, l'algorithme calcule une extension linéaire d'une orientation transitive de $\overline{G}$ quand G est un graphe de comparabilité. Une orientation transitive peut alors en être déduite en orientant les arêtes selon cette extension. En combinant ces deux résultats, on obtient facilement un algorithme de reconnaissance des graphes de permutation de même complexité (voir [57]). Cet algorithme calcule une permutation représentant le graphe à partir des deux extensions linéaires obtenues pour G et $\overline{G}$. Rappelons que les graphes de permutation sont les graphes à la fois de comparabilité et de cocomparabilité.

Reconnaissance des graphes d'intervalles

Un graphe d'intervalle est un graphe chordal dont les cliques maximales peuvent être ordonnées avec la propriété de *consécutivité*, c'est-à-dire où les cliques contenant un sommet donné sont consécutives. Un tel arrangement s'appelle une chaîne de cliques. La reconnaissance d'un graphe d'intervalle repose sur la découverte d'une chaîne de cliques. L'algorithme qui suit calcule un tel arrangement des cliques par affinage successifs de partitions de l'ensemble des cliques. Une idée similaire est utilisée par Hsu et Ma pour reconnaître les graphes d'intervalles premiers [46]. Ils utilisent ensuite un algorithme de décomposition modulaire spécifique aux graphes chordaux pour reconnaître les graphes d'intervalles quelconques. L'algorithme que nous allons aborder reconnaît n'importe quel graphe d'intervalle.

Une chaîne de cliques est calculée en affinant une partition de l'ensemble des cliques telle que les cliques contenant un sommet donné apparaissent toujours dans des boîtes consécutives. La règle d'insertion des nouvelles boîtes consiste à rassembler toutes les cliques contenant un sommet donné appelé pivot. Chaque sommet ne sert qu'une seule fois de pivot. On dira par abus de langage qu'un sommet apparaît dans une boîte si une clique maximale le contenant appartient à cette boîte. Un arbre de cliques permet pivoter sur les sommets dans un ordre adéquat (il ne faut pas pivoter sur un sommet avant qu'il n'apparaisse dans deux boîtes différentes).

Quand il n'y a plus de pivots à utiliser, chaque boîte non réduite à un singleton correspond à un module (l'ensemble des sommets qui n'apparaissent que dans cette boîte), et on sait alors d'après le théorème 40 que la dernière clique de la boîte visitée par lex-BFS est une clique extrêmale possible. En sortant cette clique de la boîte, on relance le processus d'affinage de la partition. La figure 4.8

illustre une exécution de l'algorithme sur un exemple. Voir aussi l'algorithme 4.10.

Algorithme 4.10 Affinage de partition des cliques

Données : Un graphe $G = (V, \mathcal{E})$.

Résultat : Détermine si G est un graphe d'intervalles et fournit une chaîne de cliques L si c'est le cas.

Début

1 Calculer les cliques maximales et un arbre de clique $T = (\mathcal{I}, \mathcal{F})$ avec lex-BFS d'après l'algorithme 4.8

Si *G n'est pas chordal* **Alors**

Stopper, G n'est pas un graphe d'intervalles

Soit $\mathcal{I}$ l'ensemble des cliques maximales $\mathcal{I} = \{C_1, \ldots, C_k\}$

$\mathcal{L} \leftarrow (\mathcal{I})$

$Pivots = \emptyset$ est une pile vide

Tant que $\mathcal{L} = (B_1, \ldots, B_l)$ *contient une boîte B_c non réduite à un singleton* **effectuer**

Si $Pivots = \emptyset$ **Alors**

Soit C_d la dernière clique de B_c visitée par lex-BFS (la clique de plus grand numéro).

Retirer C_d de B_c et insérer $\{C_d\}$ juste à droite de B_c.

$\mathcal{C} = \{C_d\}$

Sinon

Soit u un sommet dans $Pivots$ qui n'a pas encore servi de pivot (jeter ceux qui ont déjà servi).

Soit $\mathcal{C}$ l'ensemble des cliques contenant u.

Affiner selon $\mathcal{C}$:

2 **Si** *toutes les cliques de $\mathcal{C}$ apparaissent dans des boîtes consécutives* **Alors**

3 Soit B_a la boîte la plus à gauche contenant une telle clique.

4 Soit B_b la boîte la plus à droite contenant une telle clique

Sinon Stopper, G n'est pas un graphe d'intervalles.

5 **Si** *une boîte B_e, $a < e < b$, contient une clique qui n'est pas dans $\mathcal{C}$* **Alors** Sortir, G n'est pas un graphe d'intervalles.

Remplacer B_a par $B_a - \mathcal{C}, B_a \cap \mathcal{C}$ et B_b par $B_b \cap \mathcal{C}, B_b - \mathcal{C}$ (jeter les éventuelles boîtes vides).

Pour toute *arête C_iC_j de l'arbre de cliques reliant clique $C_i \in \mathcal{C}$ à une clique $C_j \notin \mathcal{C}$* **effectuer**

$Pivots = Pivots, C_i \cap C_j$

Retirer C_iC_j de l'arbre de cliques.

Fin

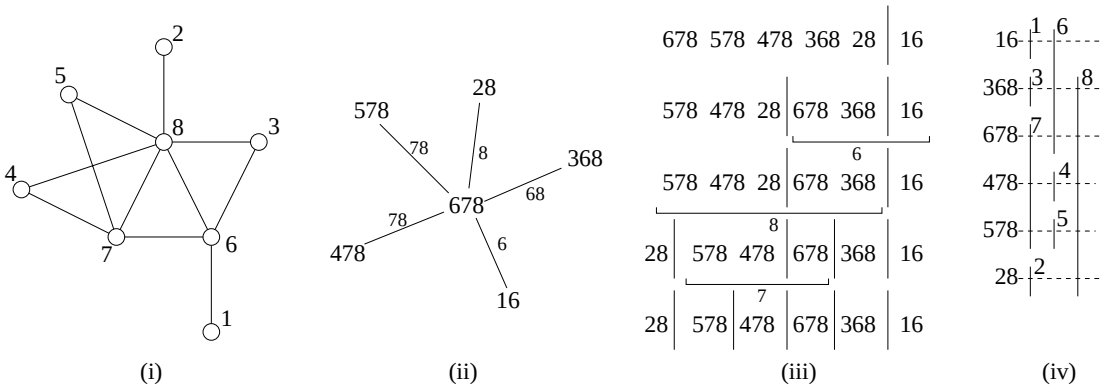


FIG. 4.8 – *(i) Un graphe d'intervalles. Ces sommets sont numéroté selon une permutation lex*-BFS. *(ii) L'arbre de cliques associé à la permutation lex*-BFS. *(iii) Affinages de partitions de l'ensemble des cliques.* $\{4, 5\}$ *est un module et la boîte* $578, 478$ *est coupée d'après le fait que que* 478 *a été découverte après* 578 *par lex*-BFS. *(iv) Une représentation intervallaire associée à la chaîne de cliques calculée. Deux sommets sont reliés si et seulement si leurs intervalles associés se recouvrent.*

Si le fait de rassembler les cliques qui contiennent un sommet donné conduit intuitivement à la propriété de consécutivité, la preuve de l'exactitude de cet algorithme découle plus naturellement d'arguments d'orientation transitive. Rappelons pourquoi les graphes d'intervalles sont les graphes chordaux de cocomparabilité. Il suffit de considérer une représentation intervallaire du graphe en ordonnant les intervalles de la gauche vers la droite pour orienter transitivement le complémentaire du graphe : si deux intervalles ne se recouvrent pas, celui qui est entièrement à gauche de l'autre est plus petit que l'autre. Le lien entre une orientation du graphe complémentaire et une chaîne de clique apparaît dans l'invariant suivant qui est conservé par l'algorithme.

Théorème 41 *Il existe une orientation transitive du graphe complémentaire telle que l'invariant suivant est vérifié : si* $\widehat{uv} \notin \mathcal{E}$ *et si* u *et* v *apparaissent dans deux boîtes différentes, alors* $\widehat{uv}$ *est orientée de la gauche vers la droite : u apparaît à gauche de v (ce qui implique que toutes les cliques contenant u apparaissent à gauche de toute clique contenant v ou dans la même boîte).*

Laissons de côté la preuve de la conservation de cet invariant pour le moment et expliquons pourquoi il implique que l'algorithme calcule une chaîne de cliques. Supposons par l'absurde que la permutation finale n'en est pas une. Soit u un sommet apparaissant à la gauche et la droite d'une clique C ne le contenant pas. Par maximalité, C doit contenir un sommet v qui n'est pas voisin de u. Cela contredit l'invariant puisque la non arête $\widehat{uv}$ ne peut être orientée à la fois par uv et vu. Toutes les cliques contenant u sont donc consécutives.

Cet algorithme ne calcule pas seulement une chaîne de cliques quand il en existe une, il détermine aussi si l'entrée est un graphe d'intervalle. A la ligne 3 (respectivement 4), B_a (respectivement B_b) est la seule boîte telle que B_{a-1} (respectivement B_{b+1}) ne contient aucune clique de $\mathcal{C}$. S'il existe une autre boîte vérifiant cela, l'invariant est violé et l'entrée ne peut pas être un graphe d'intervalle. Tout ces tests sont effectués durant le parcours de la liste des cliques maximales contenant u. Si l'entrée n'est pas un graphe, cela est donc détecté à la ligne 2 ou à la ligne 5 ou durant l'étape de reconnaissance de graphe chordal à la ligne 1. Les sommets ajoutés dans *Pivots* apparaissent toujours dans deux boîtes différentes au moins, ce qui assure $a \neq b$.

Cet algorithme est linéaire parce que chaque sommet ne sert qu'une fois de pivot, ce qui coûte $O(Degré(u))$. Le nombre de cliques contenant un sommet u donné est borné par $Degré(u)$. Les arêtes de l'arbre de clique visitées durant le pivotage sur u correspondent à des séparateurs minimaux contenant u et leur nombre est borné par $Degré(u)$. D'un autre côté, l'algorithme ne dépense pas trop de temps à jeter des sommets ayant déjà servi de pivot. En effet, le nombre total de sommets ajoutés dans *Pivots* est inférieur à m puisque les séparateurs minimaux associés aux arêtes de l'arbre de cliques sont des étiquettes lex-BFS de sommets distincts (le cardinal de l'étiquette d'un sommet est borné par son degré).

Pour prouver la conservation de l'invariant par l'algorithme, nous avons besoin du lemme suivant.

Lemme 42 *Quand tous les sommets apparaissant dans deux boîtes différentes ont servi de pivot, l'ensemble des sommets apparaissant uniquement dans une boîte donnée forment un module du graphe.*

Preuve. Soit B une boîte à un moment de l'algorithme où il n'y a plus de pivots, et soit M l'ensemble des sommets n'apparaissant que dans B. Soit u un sommet qui n'est pas dans M. Si u a voisin v dans M, une clique maximale de B les contient tous les deux. Comme u n'est pas dans M, il apparaît aussi dans une autre boîte et il a donc forcément été utilisé comme pivot. Donc toutes les cliques de B contiennent u, ce qui veut dire que tous les sommets apparaissant dans B (et par conséquent ceux de M) sont voisins de u. □

Preuve de la conservation de l'invariant. Des modules disjoint peuvent être orientés transitivement indépendamment les uns des autres et indépendamment des non arêtes sortantes. Il découle du théorème 40 et du lemme 42 que l'invariant n'est pas violé en isolant la dernière clique découverte par lex-BFS dans une boîte quand il n'y a plus de pivots.

Montrons que le pivotage sur un sommet conserve aussi l'invariant. Supposons qu'un pivot x coupe une boîte B en B_1, B_2. Il y a deux cas similaires, prouvons celui dans lequel B_1 est faite des cliques de B contenant x, l'autre cas est symétrique. Supposons $\widehat{uv} \notin \mathcal{E}$, u apparaissant à gauche de v. Il faut montrer que cette

non arête est orientée de u vers v. Il n'y a qu'un seul cas non trivial concernant les boîtes où apparaissent u et v : quand u apparaît dans B_1 et v dans B_2 (dans les autres cas, il suffit d'utiliser la conservation de l'invariant à une étape antérieure de l'algorithme).

Soit $C \in B_2$ une clique contenant v. C doit contenir un sommet z qui n'est pas voisin de x, sinon elle ne serait pas maximale. x apparaît dans une classe à gauche de B puisqu'il est utilisé comme pivot. L'invariant implique alors que la non arête entre x et z est orientée par xz. Si v et x ne sont pas voisins, on peut prendre $z = v$ et xz force uv. Dans le cas contraire, il doit y avoir une non arête entre u et z, sinon $uzvx$ induirait un cycle sans corde de longueur 4. xz force uz qui force uv. Cela prouve que l'invariant est conservé. □

Le problème de calcul d'une chaîne de clique a une traduction matricielle assez célèbre. Il est possible d'adapter les algorithmes proposés pour les graphes d'intervalles pour résoudre ce problème efficacement.

Test de la propriété des 1 consécutifs

Une matrice M de zéros et de un avec n ligne et k colonnes est dite vérifier la *propriété des 1 consécutifs* si ses colonnes peuvent être permutées de sorte que les 1 soient consécutifs dans chaque ligne. Ce problème est relié à celui de trouver une élimination de Gauss dans une matrice de réels qui n'augmente pas le nombre d'entrées non nulles [37]. Le seul algorithme jusqu'à présent permettant de résoudre ce problème utilise une structure de maniement très complexe, un PQ-tree [7], qui permet de représenter toutes les permutations des colonnes qui permettent d'obtenir la propriété des 1 consécutifs. Nous allons voir comment étendre l'algorithme 4.10 pour trouver une telle permutation grâce à la technique d'affinage de partition une fois de plus.

Définissons la *matrice sommets–cliques maximales* $M_c(G)$ d'un graphe G en associant chaque ligne à un sommet de G et chaque colonne à une clique maximale de G et où toute entrée i, j de la matrice vaut 1 lorsque le sommet associé à la ligne i appartient à la clique associée à la colonne j et 0 dans le cas contraire. Si G est un graphe d'intervalle, alors $M_c(G)$ possède clairement la propriété des 1 consécutifs. La réciproque est fausse : une matrice n'est pas toujours la matrice sommets–cliques maximales d'un graphe. Cependant, nous allons voir comment modifier une matrice M en une autre matrice $\widetilde{M}$ telle que M vérifie la propriété des 1 consécutifs si et seulement si $\widetilde{M}$ est la matrice sommets–cliques maximales d'un graphe d'intervalles. Etant donné une matrice M, définissons tout d'abord pour cela le graphe $G(M)$ de sommets les lignes de M où deux lignes sont reliées par une arête si et seulement s'il existe une colonne possédant des entrées à 1 dans les deux lignes. Les colonnes de M représentent donc des cliques de $G(M)$, mais qui ne sont pas nécessairement maximales.

Lemme 43 *Si une matrice M vérifie la propriété des 1 consécutifs, alors $G(M)$*

est un graphe d'intervalle.

Nous identifierons par la suite les colonnes de M comme des cliques de $G(M)$ pour alléger les notations (une colonne C_j sera considérée comme l'ensemble des lignes i telles que $M[i,j] = 1$).

Preuve. Nous allons montrer que M est *conformale*, c'est-à-dire que pour tout triplet de colonnes C_1, C_2, C_3, il existe toujours une colonne C contenant $(C_1 \cap C_2) \cup (C_2 \cap C_3) \cup (C_3 \cap C_1)$. Un résultat de théorie des hypergraphes permet alors de conclure [6]. Supposons que C_1, C_2, C_3 apparaissent dans cet ordre dans une permutation satisfaisant la propriété des 1 consécutifs. $(C_1 \cap C_2) \cup (C_2 \cap C_3)$ est clairement inclus dans C_2. La propriété des 1 consécutifs assure que $C_1 \cap C_3$ l'est aussi. □

Les colonnes de M ne correspondant pas toujours à cliques qui sont maximales, nous travaillerons sur la matrice $\widetilde{M}$ obtenue en rajoutant une ligne factice dans chaque colonne : chaque ligne factice contient un 1 dans la colonne qui lui est associée et des 0 dans les autres colonnes. Cette transformation ne change rien en ce qui concerne la propriété des 1 consécutifs. Comme les colonnes de $\widetilde{M}$ sont maximales, on obtient :

Théorème 44 *Une matrice M vérifie la propriété des 1 consécutifs si et seulement si $\widetilde{M}$ est la matrice sommets–cliques maximales d'un graphe d'intervalle.*

L'ajout des lignes factices n'augmentera pas sensiblement la complexité de l'algorithme 4.10 qui est linéaire en le nombre de 1, la largeur et la hauteur de la matrice d'entrée : il considère les listes de cliques contenant chaque sommet. Il reste un problème de taille : comment calculer un arbre de clique de $G(\widetilde{M})$ et vérifier si ce graphe est chordal à partir de cette représentation sous forme de liste de cliques ? En effet, on ne peut pas calculer explicitement les arêtes de ce graphes car il peut y en avoir beaucoup plus que de 1 dans la matrice. L'algorithme 4.11 permet cependant de simuler lex-BFS avec cette représentation. De manière surprenante c'est l'ensemble des cliques qu'il faut considérer, l'algorithme affine une partition des cliques jusqu'à un ordre $C_1, \ldots, C_k$ de visite des cliques par lex-BFS et une permutation lex-BFS $x_1, \ldots, x_n$ associée en même temps.

Preuve de l'algorithme 4.11. Considérons un parcours classique selon lex-BFS de $G(\widetilde{M})$ où $x_n, \ldots, x_{i+1}$ sont les sommets numérotés à un moment donné et $C_1, \ldots, C_j$ les premières cliques maximales identifiées. Si $\{x_n, \ldots, x_{i+1}\} = C_1 \cup \cdots \cup C_j$, un sommet x d'étiquette maximale appartient forcément à une clique maximale C distincte de $C_1, \ldots, C_j$. Tous les sommets non numérotés de C ont même étiquette, sans quoi l'étiquette de x contiendrait un sommet non adjacent à un des sommets non visités de C, ce qui contredirait $\{x\} \cup \mathit{Etiquette}(x) \subseteq C$. Quand x est numéroté, les sommets non numérotés de C deviennent exactement ceux d'étiquette maximale. Aussi ils seront tous visités avant les sommets qui ne sont pas dans C. Dans l'algorithme 4.11, quand une nouvelle colonne (c'est-à-dire

Algorithme 4.11 Versions matrice de lex-BFS

Données : Une matrice sommets–cliques maximales M de n lignes, k colonnes et contenant au plus m 1.

Résultat : Une permutation lex-BFS $x_1, \ldots, x_n$.

Début

  Pour tout *colonne* D **effectuer**
    $Etiquette(D) = \emptyset$
  $i \leftarrow n$
  Pour $j = 1$ *à* k **effectuer**
    Prendre une colonne D d'étiquette maximale pour l'ordre lexicographique.
    $C_j \leftarrow D$
    Tant que D *a un 1 dans une ligne non numérotée* u **effectuer**
      $x_i \leftarrow u$
      Pour toute *colonne non numérotée* D' *contenant un 1 dans la ligne* u **effectuer**
        $Etiquette(D') \leftarrow Etiquette(D'), i$
      $i \leftarrow i - 1$

Fin

une nouvelle clique maximale) est numérotée, toutes les lignes non numérotées qui ont un 1 dans cette colonne (c'est-à-dire les sommets non numérotés de la clique correspondante) sont visitées à la suite. Il suffit donc de prouver que lorsque les sommets des cliques maximales $C_1, \ldots, C_j$ sont tous numérotés, le sommet x visité ensuite par l'algorithme 4.11 peut-être choisi par un lex-BFS classique.

Montrons le par récurrence. C'est clairement vrai pour le premier sommet visité. Supposons que cela soit le cas pour l'ensemble des sommets $\{x_n, \ldots, x_{i+1}\} = C_1 \cup \cdots \cup C_j$. Soit D la colonne nouvellement visité (et qui correspont à la clique maximale C_{j+1}) et x le premier sommet choisi par l'algorithme 4.11. Dans un lex-BFS classique, l'étiquette d'un sommet y non numéroté est l'ensemble de ses voisins numérotés qui forme une clique puisque $G(\widetilde{M})$ est chordal et $x_{i+1}, \ldots, x_n$ est la fin d'un ordre d'élimination simpliciel. L'étiquette de y est donc l'étiquette des cliques maximales (colonnes) qui le contiennent lui et ses voisins numérotés. L'étiquette de x est donc l'étiquette de D qui est bien maximale. □

L'algorithme 4.12 montre comment étendre cet algorithme pour qu'il vérifie de plus si l'entrée correspond à un graphe chordal et calcule dans ce cas un arbre de cliques.

Les algorithmes 4.11 et 4.12 tournent en temps $O(n + k + m)$. En combinant ce résultat avec l'algorithme 4.10, on obtient un algorithme permettant de tester la propriété des 1 consécutifs en temps linéaire, en fournissant de plus une

Algorithme 4.12 Versions matrice du calcul de l'arbre de cliques

Données : Une matrice sommets–cliques maximales M de n lignes, k colonnes et contenant au plus m 1.

Résultat : Une permutation lex-BFS $x_1, \ldots, x_n$ et un arbre de cliques $T = (\mathcal{I}, \mathcal{F})$ sur les colonnes.

Début

  Pour toute *colonne* D **effectuer**

    $Etiquette(D) = \emptyset$

  $i \leftarrow n$

  Pour $j = 1$ *à* k **effectuer**

    Prendre une colonne D d'étiquette maximale pour l'ordre lexicographique.

    $C_j \leftarrow D$

    Relier D à $Y(Dernier(D))$: $\mathcal{F} \leftarrow \mathcal{F} \cup (D, Y(Dernier(D)))$

    Tant que D *a un 1 dans une ligne non numérotée* u **effectuer**

      $x_i \leftarrow u$

      $Y(u) \leftarrow j$

      Pour toute *colonne non numérotée* D' *contenant un 1 dans la ligne* u **effectuer**

        $Etiquette(D') \leftarrow Etiquette(D'), i$

        $Dernier(D') \leftarrow u$

      $i \leftarrow i - 1$

Fin

permutation des colonnes satisfaisant cette propriété.

Nous allons maintenant voir que la technique d'affinage est proche des idées algorithmiques utilisées en parallélisme.

4.3 Affinage de partition en parallèle

Il existe un algorithme parallèle d'affinage de partition : l'algorithme de reconnaissance des graphes chordaux de Klein qui le présente comme un algorithme basé sur une technique d'affinage de partition («*semiorder refinement* » selon ses termes). Cet algorithme est assez complexe et ne se généralise pas à d'autres problèmes. Cette section donne quelques idées sur une définition possible de l'affinage de partition en parallèle en présentant en quelques mots cet algorithme et en donnant ensuite une version parallèle de quicksort sous forme d'affinages de partition.

Reconnaissance des graphes chordaux en parallèle

lex-BFS est $\#P$-complet : [17] : il n'existe pas à priori d'algorithme parallèle calculant une permutation lex-BFS en temps polylogarithmique avec un nombre polynômial de processeurs (la classe P est un peu l'analogue de la classe NP en séquentiel). KLEIN a néanmoins trouvé un algorithme parallèle [49] permettant de calculer un ordre d'élimination simpliciel d'un graphe chordal en temps $O(\log^2 n)$ avec $n + m$ processeurs dans le modèle CRCW PRAM arbitraire. Cet algorithme consiste à couper toutes les boîtes simultanément en ne considérant pour chaque boîte que les arêtes possédant une extrémité dans la boîte. A la fin de l'algorithme, la permutation obtenue est un ordre d'élimination simpliciel. Une boîte est coupée de manière différente selon certaines propriétés sur le nombre de voisins dans la boîte des sommets de la boîte, de manière à toujours couper la boîte en sous-boîtes de taille au plus quatre cinquièmes de la taille de la boîte. Il y a en tout huit procédures différentes pour couper une boîte, cet algorithme est assez compliqué.

Quicksort

Quicksort peut être vu comme un algorithme d'affinage de partition où l'on peut faire plusieurs affinages simultanés de la partition. C'est d'un certain point de vu un algorithme parallèle d'affinages de partition. L'algorithme 4.13 le décrit ainsi.

Algorithme 4.13 [50] Quicksort

Données : Un ensemble E d'éléments totalement ordonnés par $<_{tot}$.
Résultat : Une liste $\mathcal{L}$ des éléments triés selon $<_{tot}$.
Début
  $\mathcal{L} \leftarrow (E)$
  Tant que $\mathcal{L} = (B_1, \ldots, B_k)$ *contient une boîte non réduite à un singleton* **effectuer**
1   **Pour toute** *boîte* B_i *non réduite à un singleton* **effectuer**
2   Prendre un élément $u \in B_i$.
      Affiner $\mathcal{L}$ selon l'ensemble S_u des éléments supérieurs à u.
      Affiner $\mathcal{L}$ selon l'ensemble I_u des éléments inférieurs à u.
Fin

L'exactitude de l'algorithme provient de la conservation de l'invariant suivant : « si deux éléments v et w sont dans deux boîtes distinctes B_a et B_b respectivement alors $v <_{tot} w \iff a < b$ ». (Si v et w ont étés séparés par un pivot u, alors $v <_{tot} u <_{tot} w$ et $a < b$ ou bien $w <_{tot} u <_{tot} v$ et $b < a$.) La boucle de la ligne 1 peut être parallélisée car l'invariant implique que les deux affinages selon un pivot $u \in B_i$ ne morcellent que la boîte B_i. Pour faire ces deux affinages il suffit donc de calculer $S_u \cap B_i$ et $I_u \cap B_i$ en comparant selon $<_{tot}$ chaque élément dans B_i

avec u. Si u est choisi aléatoirement à la ligne 2, l'algorithme termine avec forte probabilité en $O(\log n)$ affinages parallèles[8]. Cela représente un travail total en $O(nT_{<_{tot}} \log n)$ avec forte probabilité où $T_{<_{tot}}$ est le travail maximal nécessaire à la comparaison de deux éléments selon $<_{tot}$.

S'il y a des éléments égaux à u, ils se retrouvent dans la même boîte que u après avoir pivoté sur u ; cette boîte n'est constituée que des éléments égaux à u et ne peut plus être morcelée par la suite. La condition d'arrêt de l'algorithme devient : « chaque boîte a été obtenue comme l'ensemble des éléments égaux à un pivot ».

Si $<_{tot}$ n'est pas un ordre total, cet algorithme calcule une extension linéaire de cet ordre quand les sommets incomparables à u sont traités comme s'ils étaient égaux à u. En revanche, la boîte contenant u doit encore être morcelée par la suite (en évitant de pivoter deux fois sur le même élément). L'invariant est alors le suivant : « si deux éléments u et v sont dans deux boîtes distinctes B_a et B_b respectivement alors $u <_{tot} v \Longrightarrow a < b$ ». (Si v et w, avec $v <_{tot} w$, ont étés séparés par un pivot u, alors au moins l'un des deux éléments est comparable à u. S'ils le sont tous les deux, alors $v <_{tot} u <_{tot} w$ et $a < b$. Si u est comparable à v mais pas à w, alors $v <_{tot} u$ car $u <_{tot} v$ impliquerait que u et w sont comparables et on a alors $a < b$. Si u est comparable à w mais pas à v, alors $u <_{tot} w$ et on a donc $a < b$ dans tous les cas.) La condition d'arrêt de l'algorithme devient : « chaque boîte contient au plus un sommet qui n'a pas servi de pivot ». A la fin de l'algorithme, les boîtes sont des *antichaînes*, c'est-à-dire des ensembles d'éléments deux à deux incomparables. La boucle de la ligne 1 peut toujours être parallélisée.

Malheureusement, les résultats en complexité sont plus faibles : si $<_{tot}$ est un ordre « vide » où tous les éléments sont incomparables, il faut n rondes d'affinages pour se rendre compte que E est une antichaîne. Le travail total de l'algorithme est donc en $O(T_{<_{tot}} n^2)$ où $T_{<_{tot}}$ est le temps nécessaire à la comparaison de deux éléments selon $<_{tot}$. Cela laisse tout de même un peu d'espoir en ce qui concerne la conception d'un algorithme parallèle efficace de calcul d'une extension linéaire. Remarquons pour cela que l'on est pas obligé d'affiner la partition selon S_u et selon I_u à la suite, on peut s'autoriser à pivoter au plus deux fois sur chaque sommet : une fois en affinant selon S_u, une fois en affinant selon I_u (cela peut couper la boîte de u en deux parties comparables dans un cas et pas dans l'autre). Une stratégie permettant de résoudre le problème d'un ordre « vide » (ou presque vide) pourrait consister à calculer après chaque ronde d'affinages les degrés entrants et sortants de chaque sommet à l'intérieur de sa boîte.

Dans le cas où $<_{tot}$ est donné par des listes de successeurs (il est facile d'en déduire les listes de prédécesseurs), le travail total est en $O(n + m)$ puisqu'on pivote au plus une fois par sommet.

La définition de technique d'affinage en parallèle n'est peut être pas encore assez précise pour donner ses fruits. L'idée de base est pourtant proche du paral-

8. C'est à dire en $O(c \log n)$ affinages parallèles avec probabilité supérieure à $1 - 1/n^c$.

lélisme car elle repose sur l'utilisation d'une information locale : le voisinage d'un sommet. En parallèle, il faut de plus couper toutes les boîtes en même temps. Dans les algorithmes abordés dans cette section, on a effectué en parallèle des affinages qui ne coupent qu'une boîte, on pourrait aussi combiner cela avec des affinages qui coupent toutes les boîtes (le voisinage d'un sommet de degré de l'ordre $n/2$ par exemple). Dans tous les cas, le plus difficile consiste à s'arranger pour que la taille maximale des boîtes après affinage soit au plus une fraction de la taille maximale des boîtes avant affinage. Remarquons que cette technique est de toute manière intéressante pour les modèles distribués tels que CGM. Si l'on sait couper les grandes boîtes, on peut les traiter en séquentiel dès que leur taille est inférieur à n/p. Le problème vient alors de la répartition des arcs comme dans le paragraphe 2.5

4.4 Problèmes ouverts

lex-BFS et coloriage d'un graphe

Colorier un graphe consiste à assigner une couleur à chaque sommet de sorte que deux sommets voisins ne soient pas de la même couleur. Calculer un coloriage avec un nombre de couleurs minimal est un problème difficile en général. Il existe des algorithmes efficaces pour de nombreuses classes de graphes particulières (surtout pour des sous-classes de graphes parfaits [37]). C'est le cas pour les graphes de comparabilité et les graphes chordaux.

lex-BFS peut être vu comme un algorithme de coloriage si l'on modifie la façon dont sont numérotés les sommets : quand un sommet est visité, numérotons le par le plus petit numéro qui n'est pas dans sa marque. (il marque alors ses voisins non encore visité avec ce numéro). On obtient clairement une coloration du graphe $G = (V, \mathcal{E})$ en entrée de cette manière. De plus, le nombre total de couleurs (de numéros) utilisées est borné par $\max_{u \in V} |Etiquette(u)| + 1$ (ce qui est inférieur au degré maximal plus un). Dans le cas des graphes chordaux, ce nombre est la taille de la plus grande clique et est donc le nombre de couleur minimal permettant de colorier le graphe.

En ce qui concerne les graphes quelconques, il est peut être possible de d'abaisser la borne par le degré plus un avec une règle du type « pivoter de préférence sur les sommets de grand degré »...

Décomposition modulaire

Les algorithmes linéaires de décomposition modulaire [15, 57, 19] fonctionnent sur le même principe que l'algorithme 4.4 avec l'ajout d'une structure de données assez complexe permettant de pivoter de manière plus efficace sur les sommets et en travaillant sur une représentation affaiblie des modules : une structure basée sur un P_4 (un chemin sans corde à quatre sommets) dans [15], un «P_4 *tree* » dans

[57], deux arbres de décomposition modulaire calculés récursivement dans [19]. Ces structures peuvent être calculées en pivotant un nombre constant de fois sur chaque sommet. La partie complexe de ces algorithmes consiste à calculer l'arbre de décomposition modulaire à partir de la structure plus faible.

L'algorithme 4.4, si une boîte B contient un module M avant affinage selon un sommet extérieur à B, M est forcément inclus dans une des deux boîtes provenant de B après affinage (cela découle de la définition d'un module). Quand on a de plus pivoté sur tous les sommets extérieurs à une boîte, cette boîte est un module. Ces deux principes sont souvent à la base des algorithmes de décomposition modulaire.

Un résultat théorique [14] de Cournier et Habib indique qu'il est peut-être possible de calculer directement l'arbre de décomposition modulaire sans passer par une structure intermédiaire : il existe toujours un ordre dans lequel pivoter sur les sommets d'un graphe premier (à la manière de l'algorithme 4.4) de sorte que toutes les boîtes soient finalement des singletons. L'algorithme 4.4 utilise une méthode astucieuse pour borner le nombre de pivotage à $\log n$ pour chaque sommet. Le principal problème pour concevoir un algorithme de décomposition linéaire (ou même d'orientation transitive) est de trouver un tel ordre de pivotage.

Un autre résultat intéressant obtenu par Capelle et Habib [9] montre que l'arbre de décomposition modulaire peut être calculé en temps linéaire à partir d'une permutation des sommets telle que les sommets de chaque module fort apparaissent consécutivement. L'algorithme 4.9 proposé dans le paragraphe 4.2 permet de calculer une telle permutation grâce à la technique d'affinage de partition. Je pense que le problème algorithmique de la décomposition modulaire n'est pas encore fermé (le fait qu'un quatrième algorithme linéaire de décomposition modulaire paraît bientôt [19] en témoigne). Le seul maillon qui manque est une méthode pour pivoter sur les sommets dans un ordre adéquat.

L'algorithme de reconnaissance des graphes d'intervalles proposé dans la section 4.2 qui est d'une certaine manière un algorithme d'orientation transitive du graphe complémentaire (les graphes d'intervalles sont les graphes chordaux de cocomparabilité), utilise un « arbre de cliques » calculé par lex-BFS pour ne pivoter qu'une fois au plus sur chaque sommet. Cet arbre de clique est une structure propre à tout graphe chordal (et ne constitue pas dans cet algorithme une structure de données raffinée en un arbre de décomposition). Il ne sert qu'à trouver un ordre adéquat dans lequel pivoter sur les sommets.

L'existence d'algorithmes linéaires de décomposition modulaire incite à penser qu'il existe une structure similaire pour un graphe quelconque (mis à part l'arbre de décomposition modulaire lui-même bien-sûr).

Conclusion

Nous avons vu comment la technique d'affinage de partition permet de traduire de nombreux algorithmes allant du tri au test de consécutivité des 1 d'une matrice, en passant par l'orientation transitive. Cette technique pourrait certainement permettre de reconnaître d'autres classes de graphes telles que les graphes de permutation, les graphes trapézoïdaux, les graphes faiblement chordaux, ... et même des classes intéressantes de graphes bipartis. De plus, quand une classe de graphes peut être reconnue en utilisant cette technique, alors la classe complémentaire (des graphes dont le complémentaire est dans cette classe) aussi. Les algorithmes de calcul de plus courts chemins à partir d'une source utilisent des techniques qui pourraient fort bien s'exprimer en termes d'affinage de partition. Nous avons aussi vu comment ce passer des PQ-trees pour tester de manière plus simple la propriété des 1 consécutifs dans une matrice. Les PQ-trees servent aussi à reconnaître les graphes planaires, la technique d'affinage pourrait s'appliquer à ce problème aussi.

Conclusion

L'algorithmique des graphes est très riche et cette thèse ne donne finalement qu'un aperçu assez réduit de l'éventail varié des diverses techniques existantes. L'intérêt du travail effectué ici réside surtout dans l'étude de certains problèmes dans divers modèles de calcul : séquentiel parallèle à grain fin et parallèle à gros grain en passant par un problème particulier de réseau de téléphonie mobile qui constitue une sorte de modèle de calcul soumis à de très nombreuses contraintes.

Tout ce travail s'articule autour de la recherche de techniques générales car la grande variété de techniques algorithmiques existante est aussi un frein à leur compréhension par les programmeurs qui ne sont pas familiers du domaine. En parallèle, nous avons pu nous rendre compte que le tri et le calcul de composantes connexes sont malheureusement pratiquement les seuls outils assez généraux permettant de traiter les graphes de manière efficace. Pour de nombreux problèmes élémentaires en algorithmique des graphes la seule solution existante est le produit de matrice (ou la fermeture transitive), ce qui ne constitue qu'une solution partielle car certains graphes (les graphes « creux ») peuvent présenter une matrice ou une fermeture transitive trop volumineuse pour tenir dans la mémoire d'un ordinateur, même massivement parallèle alors que leur représentation par listes d'adjacence est nettement plus petite.

Avec la technique d'affinage de partitions, j'ose espérer que nous avons permis de dégager un filon d'algorithmes efficaces en montrant une manière facile à comprendre de les appréhender. J'ai encore l'espoir d'appliquer cette technique dans des modèles parallèles de calcul, même si mon expérience me dit que c'est une tâche difficile. Dans le cas distribué, cette façon de partitionner me paraît judicieuse car distribuer efficacement sur des mémoires distinctes une structure irrégulière telle qu'un graphe constitue un problème difficile. Les algorithmes qui peuvent se déduire en termes d'affinage de partition offrent une direction naturelle pour résoudre ce problème.

Je voudrais encore souligner la richesse de raisonner dans différents modèles de calcul. Ma vision de l'affinage de partition a très certainement bénéficié de mon expérience en parallélisme. D'une manière générale, j'aimerais aussi confronter l'algorithmique à d'autres domaines. L'algorithmique pourrait s'appliquer dans des domaines extérieurs à l'informatique comme l'organisation d'un chantier ou d'une grande entreprise.

Je considère l'algorithmique comme une science très ludique car elle permet de réaliser certains raisonnements de l'esprit humain. On peut bien sûr étendre cette remarque en ce qui concerne la programmation qui en est le laboratoire expérimental. Les ordinateurs ont rendu si facile la réalisation de méthodes que le premier réflexe d'un programmeur en face d'un problème est d'essayer un bout de code et non d'aller voir dans les livres d'algorithmique s'il existe déjà une solution. L'algorithmique a subi un développement presque anarchique où l'on a essayé de résoudre tous les problèmes auxquels on était confronté. Je pense que l'algorithmique doit s'attacher à dégager des lignes générales de résolution de problèmes, on ne peut donner une solution à chacune des infinies instances de problèmes. Il me semble que l'algorithmique ne sert pas assez au programmeur. La seule technique un peu évoluée implantée par défaut dans la plupart des langages est le tri, un des plus vieux algorithmes de l'ère des ordinateurs. L'évolution des langages de programmation commence à rendre possible l'utilisation d'un morceaux de programme écrit par une personne à d'autres personnes. L'idéal serait de pouvoir réemployer un algorithme écrit et déjà implanté par une tierce personne (même pour une utilisation différente), c'est-à-dire faire quelque chose qui s'avère très difficile à l'heure actuelle : modifier un programme existant (ou une fonction de ce programme) pour une utilisation similaire mais différente. La théorie de la complexité est une des premières réussites de cette jeune science qu'est l'informatique de l'ère des ordinateurs. J'espère que cette thèse contribuera par son contenu un tant soit peu à faire un jour de l'algorithmique une science plus aboutie et plus abordable au profane.

Bibliographie

[1] E.H.L. Aaarts and P.J.M. Van Laarhoven. *Simulated Annealing: Theory and Applications*. New York Wiley, 1987.

[2] R. Azencott. *Simulated annealing: parallelization techniques*. New York Wiley, 1992.

[3] K.A. Baker, P.C. Fishburn, and F.S. Roberts. Partial orders of dimension 2. *Networks*, 2:11–28, 1971.

[4] K.E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conf.*, pages 307–314, 1968.

[5] M. Bender, M. Gastaldo, and M. Morvan. Parallel interval order recognition and construction of interval representation. Technical Report RR 93-27, LIP ENS-Lyon, September 1993. To appear in Theoretical Computer Science.

[6] C. Berge. *Graphes et Hypergraphes*. Dunod, 1970.

[7] 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.

[8] C. Capelle. Dimension and algorithms. In V. Bouchité and M. Morvan, editors, *Orders, Algorithms, and Applications*, number 831 in Lect. Notes Comput. Sci., pages 143–161. Springer-Verlag, 1994.

[9] C. Capelle and M. Habib. Graph decompositions and factorizing permutations. Technical report, LIRM Montpellier, 1996. soumis.

[10] O. Catoni. Applications of sharp large deviation estimates to optimal cooling schedules. *Ann. Inst. Henri Poincaré, Probalités et statistiques*, 27(4), 1991.

[11] C.J. Colbourn. On testing isomorphism of permutation graphs. *Networks*, 11:13–21, 1981.

[12] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[13] R. Cole and U. Vishkin. Approximate paralle scheduling. application to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92:1–47, 1991.

[14] A. Cournier and M. Habib. An efficient algorithm to recognize prime undirected graphs. In *WG'92 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 657 of *Lecture Notes in Computer Science*, pages 212–224. 21st Internation-

nal Workshop WG'95, Springer-Verlag, 1992.

[15] 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-Verlag.

[16] R. Cypher and C.G. Plaxton. Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers. In *22nd Annual Symposium on Theory of Computing*, pages 193–203, October 1990.

[17] E. Dalhaus. The parallel complexity of elimination ordering procedures. In J. van Leeuwen (Ed.), editor, *Proc. 19th International Workshop on Graph Theoretic Concepts in Computer Science*, volume 790 of *LNCS*, pages 225–, Utrecht, The Netherlands, June 1993. WG'93, Springer Verlag, Berlin, 1994.

[18] E. Dalhaus. Efficient parallel modular decomposition. In *WG '95 21st International Workshop on Graph-Theoretic Concepts in Computer Science*. M. Nagl, 1995. To appear in Lecture Notes in Computer Science.

[19] E. Dalhaus, J. Gustedt, and R.M. McConnell. Efficient and practical modular decomposition. In *Proceedings of the seventh annual ACM-SIAM Symposium on Discrete Algorithm*. Society of Industrial and Applied Mathematics (SIAM), 1997.

[20] F. Dehne, A. Fabri, and C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*. Dallas USA, October 1994.

[21] A. Denise. *Méthodes de génération aléatoire d'objets combinatoires de grande taille et problèmes d'énumeration*. PhD thesis, Universite Bordeaux I, janvier 1994.

[22] P.F. Dietz. Optimal algorithms for list indexing and subset rank. In *Algorithms and data structures, Proc. workshop WADS '89, Ottawa/Canada*, number 382 in Lect. Notes Comput. Sci., pages 39–86, 1989.

[23] J.-L. Dornstetter. Nortel matra. Communication privée, 1996.

[24] A. Ehrenfeught and G. Rozenberg. Theory of 2-structures. *Theoretical Computer Science*, 70:277–342, 1990.

[25] S.E. Elmaghraby. *Activity Networks*. Wiley, New York, 1977.

[26] A. Fabri, F. Dehne, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. of the 9th ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[27] U. Faigle, G. Gierz, and R. Schrader. Algorithmic approaches to setup minimization. *SIAM J. Comput.*, 14:954–965, 1985.

[28] A. Ferreira. Halifax canada. Communication privée, juillet 1997.

[29] A. Ferreira, F. Dehne, A. Rau-Chaplin, and I. Computing connected components and list-ranking on coarse grained multi-computers. non publié.

[30] A. Ferreira, C. Kenyon, A. Rau-Chaplin, and S. Ubéda. *d*-dimensional range search on multicomputers. Technical Report 96-23, LIP ENS-Lyon, august 1996.

[31] A. Ferreira, A. Rau-Chaplin, and S. Ubéda. Scalable 2d convex hull and triangulation for coarse grained multicomputers. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, 1996. Voir la thèse de M. Diallo, LIP ENS-Lyon, pour des résultats d'implantations.

[32] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *21st ACM STOC*, pages 345–354, 1989.

[33] 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-Verlag.

[34] T. Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Scient. Hung. Tom.*, 18:25–66, 1967.

[35] F. Gavril. The intersection graphs of a path in a tree are exactly the chordal graphs. *Journ. Comb. Theory*, 16:47–56, 1974.

[36] D. Geman and S. Geman. Stochastic relaxation, Gibbs distribution, Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6:721–741, 1984.

[37] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[38] T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the 28th annual ACM Symposium on Theory of Computing (STOC)*. Philadelphia USA, 1996.

[39] P. Grillet. Maximal chains and antichains. *Fund. Math.*, 65:157–167, 1969.

[40] M. Habib and R. Jegou. N-free posets as generalizations of series-parallel posets. *Discrete Appl. Math.*, 12(3):279–291, 1985.

[41] M. Habib and R.H. Möhring. Treewidth of cocomparability graphs and a new order-theoric parameter. Technical report, LIRMM, October 1992.

[42] T. Hagerup. Towards optimal parallel bucket sorting. *Information and Computation*, 75(1):39–51, octobre 1987.

[43] S.E. Hambausch and A.A. Khokar. C^3: An architecture-independent model for coarse-grained parallel machines. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*. Dallas USA, October 1994.

[44] F. Harary and Z.R. Norman. Some properties of line digraphs. *Rend. Circ. Math.*, 9(161-168), 1960. Palermo.

[45] R.L. Hemminger and L.W. Beineke. Line graphs and line digraphs. In L. W. Beineke and R. J. Wilson, editors, *Selected topics in*

graph theory, pages 271–305, London, 1978. Academic Press.

[46] 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.

[47] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. Algorithms and Complexity, pages 870 – 941. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[48] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–686, 1983.

[49] P. Klein. *Synthesis of parallel algorithms*, pages 341–408. Morgan Kaufmann Publishers, 1993. Parallel algorithms for chordal graphs.

[50] D.E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Co., 1968.

[51] D. Krob. Paris jussieu. Communication privée, février 1997.

[52] C. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. In *Internat. Conf. Parallel Processing*, pages 180–184, 1985.

[53] B. Leclerc and B. Monjardet. Orders "c.a.c.". *Fund. Math.*, 79:11–22, 1973.

[54] F.T. Leighton. Tight bounds on the complexity of sorting. *IEEE Trans. Comput.*, C-34(4):344–354, 1985.

[55] T.H. Ma and J. Spinrad. Transitive closure for restricted classes of partial orders. *Order*, 8(2):175–183, 1991.

[56] J.M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. In *Proceedings of the Allerton Conference on Computing, Communication and Control*, pages 603–612, 1987.

[57] M.R. McConnell and J. 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.

[58] 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.

[59] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Inf.*, 21:339–374, 1984.

[60] G.L. Miller, V. Ramachadran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, 17(4):687–695, 1988.

[61] J.J. Moder and C.R. Phillips. *Project Management with CPM and PERT*. Reinhold, New York, 1964.

[62] R.H. Mohring. Computationally tractable classes of ordered sets. In I. Rival, editor, *Algorithms and Order*, pages 105–193. Kluwer Acad. Publ., Dordrecht, 1989.

[63] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[64] D. Nassimi and S. Sahni. Data broadcasting in simd computers. *IEEE Trans on Computers*, 30(2):101–107, 1981.

[65] A. Pneuili, A Lempel, and W. Even. Transitive orientation of graphs and identification of permutation graphs. *Canad. J. math*, 23:160–175, 1971.

[66] F.P. Preparata and M.I. Shamos. *Range-searching problems*. Springer-Verlag, 1985. Chapter 3, 67-88.

[67] P. Radge, F. Fish, and A. Widgerson. Relations between concurrent write models of parallel computation. *SIAM Journal on Computing*, 17:606–627, 1998.

[68] S. Rajasekaran and J.H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[69] J.H. Reif. *Synthesis of parallel algorithms*. Morgan Kaufmann Publishers, 1992.

[70] I. Rival. Optimal linear extensions by interchanging chains. *Proc. Amer. Math. Soc.*, 89:387–394, 1982.

[71] D.J. Rose, R.E. Tarjan, and G.S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal of Computing*, 5(2):266–283, June 1976.

[72] J. Siarry and G. Dreyfus. *La méthode du recuit simulé*. Paris IDSET, 1989.

[73] J. Spinrad. Graph partitionning.

[74] J. Spinrad. The minimum dummy task problem. *Networks*, 16(3):331–348, 1986.

[75] J. Spinrad. Dimension and algorithms. In V. Bouchité and M. Morvan, editors, *Orders, Algorithms, and Applications*, number 831 in Lect. Notes Comput. Sci., pages 33–52. Springer-Verlag, 1994.

[76] J. Spinrad and J. Valdes. Recognition and isomorphism of two dimensional partial orders. In 10th Coll. on Automata, Language and Programming. Lecture Notes in Computer Science , volume 154, pages 676–686. Springer-Verlag, Berlin, 1983.

[77] M.M. Syslo. A labeling algorithm to recognize a line digraph and output its root graph. *Inform. Processing Letters*, 15:241–260, 1982.

[78] M.M. Syslo. On the computational complexity of the minimum-dummy-activities problem in a pert network. *Networks*, 14:37–45, 1984.

[79] M.M. Syslo. A graph theoretic approach to the jump number problem. In I. Rival, editor, *Graphs and Order*, pages 185–215, Reidel, Dordrecht, 1985.

[80] R.E. Tarjan. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14:862–874, 1985.

[81] R.E. Tarjan and J. Paige. Three partition refinement algorithms. *SIAM J. Computing*, 16(6), 1987.

[82] J.D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991.

[83] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recognition of series-parallel digraphs. *SIAM J. Comput.*, 11:298–314, 1982.

[84] L.G. Valiant. Bulk-synchronous parallel computers. In M. Reeve and S. E. Zenith, editors, *Parallel Processing and Artificial Intelligence*, pages 15–22. Wiley, 1989.

[85] L.G. Valiant. A bridging model for parallel computation. *Communication of ACM*, 38(8):103–111, 1990.

[86] L.G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Computing*, 12(4):641–644, 1983.

Index