NNT : 2011EVRY0012

École Doctorale Sciences & Ingénierie

Université d'Évry-Val d'Essonne
U.F.R. Sciences Fondamentales et Appliquées

Thèse

présentée par

Romain Campigotto

pour obtenir

le grade de Docteur en Sciences
de l'Université d'Évry-Val d'Essonne

Spécialité : informatique

Algorithmes d'approximation à mémoire limitée pour le traitement de grands graphes

– Le problème du Vertex Cover –

Soutenue le *6 décembre 2011* devant le jury composé de

Président du jury	M. Matthieu Latapy	Directeur de Recherche CNRS au LIP6
Rapporteur	M. Dominique Barth	Professeur à l'Université de Versailles
Rapporteur	M. Pascal Berthomé	Professeur à l'ENSI de Bourges
Examinateur	M. Bruno Escoffier	Maître de Conférences à l'Université Paris-Dauphine
Co-directeur de thèse	M. Christian Laforest	Professeur à l'Université Blaise Pascal
Directeur de thèse	M. Eric Angel	Professeur à l'Université d'Évry-Val d'Essonne

Thèse préparée au sein de l'équipe OPAL du Laboratoire IBISC
EA 4526 – Université d'Évry-Val d'Essonne

Remerciements

Même si ce document n'est attribué qu'à un seul auteur, cette thèse ne serait pas ce qu'elle est sans l'aide, le soutien, la présence et l'apport de nombreuses personnes. Je remercie donc l'ensemble des personnes qui ont contribué directement ou indirectement à l'accomplissement de ce travail.

Une étape importante et régulière dans la vie d'un chercheur est l'évaluation (et de manière plus générale *l'échange*). Dans ce sens, je tiens à remercier tout particulièrement les membres du jury, qui ont accepté d'évaluer mes travaux.

- Je remercie M. Matthieu Latapy, Directeur de Recherche CNRS au LIP6, de m'avoir fait l'honneur de présider le jury de cette thèse.
- Je remercie également MM. Dominique Barth, Professeur à l'Université de Versailles, et Pascal Berthomé, Professeur à l'ENSI de Bourges, d'avoir en outre accepté la charge de rapporter ce travail. Leurs remarques et suggestions m'ont notamment permis d'améliorer la qualité de ce mémoire.
- Je remercie aussi M. Bruno Escoffier, Maître de Conférences à l'Université Paris-Dauphine, d'avoir accepté le rôle d'examinateur.

Je remercie évidemment et chaleureusement mes directeurs de thèse, MM. Eric Angel et Christian Laforest, pour la très grande qualité de leur encadrement. Ce travail ne serait pas ce qu'il est sans (dans le désordre, et surtout de manière non exhaustive) : leur expérience, leurs compétences, leur disponibilité, leur soutien, leur tenacité, leur gentillesse, leur exigeance, leurs encouragements et leurs remarques (toujours judicieuses). Ils m'ont guidé de la meilleure façon qui soit durant ces trois années. De plus, ils ont toujours su faire preuve d'une complémentarité et d'une coordination sans faille (et ce malgré la distance géographique entre Évry et Clermont-Ferrand). C'est bien simple, je n'arrive toujours pas à croire que je sois le premier doctorant qu'ils co-encadrent ! En tout cas, j'espère que nous pourrons continuer à travailler ensemble le plus longtemps possible.

Depuis le début de mon stage de M2 recherche en mars 2008, j'ai passé un peu plus de trois ans au sein du Laboratoire IBISC. Je tiens à remercier chaleureusement l'ensemble des personnes que j'ai pu y cotoyer, notamment :

– pour les nombreuses discussions intéressantes et les bons moments passés en salle café, avec

une pensée émue pour le superbe gâteau qui avait été offert aux biologistes et qui s'est métamorphosé « tout seul » dans le réfrigérateur...
– l'ensemble des doctorants (et des postdocs) pour l'entraide mutuelle, les conseils avisés et (surtout) pour les moments de détente réguliers (mais pas trop longs) ;
– en particulier Laurent, Stéphane et François, pour avoir rempli de façon admirable le rôle de « grands frères de thèse » ;
– les membres de `Tech-Staff`[1] : Laurent, Pascal, Franck et Florent (entre autres), avec qui j'ai beaucoup appris sur le réseau (mais pas seulement) ;
– sans oublier l'une des pièces essentielles de ce laboratoire : Dominique (« notre mère à tous », comme le dit si bien Marie), pour sa gentillesse et son efficacité permanentes.

Je remercie aussi l'ensemble des personnes du Département Informatique de l'Université d'Évry avec qui j'ai pu travailler durant mes trois années (enrichissantes) de monitorat.

Depuis la rentrée 2011, je suis ATER à l'Université Paris-Dauphine. Je remercie les membres du LAMSADE qui ont grandement facilité mon intégration là-bas, notamment les gens du 2ème (et du 4ème), les occupants des bureaux C605 et C603bis, sans oublier bien sûr Vangelis.

Je remercie aussi mes amis et ma famille au sens large. Vous savez bien pourquoi !

Enfin, je n'aurais jamais pu en arriver là sans mes parents, qui m'ont permis d'effectuer mes études en toute sereinité et qui ont toujours su faire passer mes intérêts avant les leurs. Je suis parfaitement conscient que je leur dois beaucoup !

MERCI !

[1] Les administrateurs systèmes et réseaux, pour la plupart volontaires et bénévoles, du Laboratoire IBISC.

Table des matières

Table des figures

Liste des tableaux

Présentation générale

Depuis ces dernières décennies, l'informatique (au sens large[2]) joue un rôle de plus en plus important dans la vie de tous les jours, au point de devenir à l'heure actuelle omniprésente, voire indispensable.

Ce phénomène entraîne tout naturellement un accroissement des quantités de données informatisées, ce qui nous oblige à considérer certains problèmes sous un jour nouveau (et ce en dépit des évolutions technologiques récentes). Notamment, l'étude des problèmes d'optimisation combinatoire (qui constitue une branche importante de la recherche en informatique) voit apparaître de nouvelles problématiques directement liées à cette augmentation.

L'étude d'un problème d'optimisation peut s'effectuer sous différents angles, de façon plus ou moins théorique. Si l'on prend en compte le constat évoqué dans le paragraphe précédent, il s'agit de concevoir des méthodes de résolution qui fournissent des solutions qui sont à la fois les plus proches possible de l'optimal et qui, de plus, répondent aux problématiques liées au traitement de grandes masses de données. C'est ce que nous avons tenté de réaliser durant cette thèse.

Cette thèse porte sur le traitement d'un problème d'optimisation sur des graphes (à savoir le problème du VERTEX COVER) dans un contexte bien particulier : celui des grandes instances de données. Nous nous sommes intéressés à plusieurs algorithmes adaptés à ce contexte. Nous avons étudié la qualité de leurs solutions ainsi que leurs complexités, tout d'abord de façon théorique, puis de manière expérimentale.

A noter que cette thèse s'inscrit dans le cadre du projet ANR *ToDo*[3] *(Time versus Optimality in Discrete Optimization)*, plus particulièrement dans la tâche 3 de ce projet, intitulée « Very Fast Approximation », ainsi que dans la tâche 4, intitulée « Implementation of the Algorithms ».

Dans ce qui suit, nous décrivons l'organisation globale de ce document, puis nous donnons quelques remarques générales sur sa lecture.

[2]Comme l'ont si bien dit *M. R. Fellows et I. Parberry* dans leur article paru dans *Computing Research News* en 1993 et intitulé *SIGACT Trying to Get Children Excited About CS* : « l'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes ».

[3]Ce projet ANR réunit des chercheurs du LAMSADE, du LIMOS, de l'ESSEC et de l'IBISC. Les informations détaillées sur ce projet sont disponibles à l'adresse suivante : `http://todo.lamsade.dauphine.fr`.

Plan de la thèse

Le chapitre 1 décrit le domaine d'étude des travaux effectués durant cette thèse. Il présente le contexte, les motivations et les problématiques liées au traitement de grandes instances de données. Il compare ensuite différents modèles précédemment proposés dans la littérature et liés à ce contexte.

Il introduit par la suite le problème que nous avons étudié : le VERTEX COVER. Il montre une borne minimale sur la quantité de mémoire nécessaire pour le résoudre. Il donne ensuite un exemple d'application détaillé sur des données de grande taille, puis il effectue un tour d'horizon des principaux algorithmes existants.

Le chapitre 2 est consacré à l'étude de trois algorithmes : LL, SLL et ASLL. Ces trois algorithmes, que nous avons qualifiés de *memory-efficient*, utilisent une taille mémoire constante par rapport à la taille des données traitées. Ils sont donc adaptés au traitement des grandes instances. Nous analysons la qualité des solutions qu'ils produisent ainsi que leurs complexités, en moyenne sur des graphes généraux ainsi que sur les arbres.

Le chapitre 3 est consacré à l'étude d'autres algorithmes pour le problème du VERTEX COVER sur de grands graphes. Ces algorithmes utilisent davantage de mémoire que les algorithmes du chapitre précédent. Nous étudions leurs complexités et nous nous intéressons plus particulièrement à la quantité de mémoire utilisée par l'un d'entre eux.

Le chapitre 4 est consacré à l'étude expérimentale sur de gros graphes des algorithmes présentés et étudiés dans ce document. Il détaille les caractéristiques des expériences menées, puis il présente les résultats obtenus, à savoir la qualité des solutions construites par les algorithmes, leurs complexités ainsi que leurs durées d'exécution.

Guide de lecture

Nous supposons dans ce document que le lecteur est familier avec les notions de graphe, d'algorithme, de complexité, d'optimisation et d'approximation. On peut toutefois citer [125] pour la théorie des graphes, [43] pour l'algorithmique, ainsi que [20], [103] et [118] pour la théorie de la complexité, l'optimisation et l'approximation.

Dans ce mémoire, nous n'avons volontairement pas traduit certains termes, parce qu'ils font référence à des choses bien spécifiques et qu'ils sont bien souvent employés tel quel par la communauté française, et aussi parce que leurs traductions feraient perdre leurs significations particulières.

Dans les graphes illustrés par les figures de ce document, les sommets sont par défaut noirs ; ceux mis en évidence sont blancs, voire gris. Autrement, une légende est spécifiée.

Chapitre 1

Traiter des données de grande taille : pourquoi et comment ?

Nous allons décrire dans ce chapitre le domaine d'étude des travaux présentés dans ce document. Nous introduirons aussi les éléments qui seront utilisés tout au long de ce mémoire.

Nous donnerons dans un premier temps le contexte, les motivations et les problématiques liées au traitement de grandes instances de données. Nous discuterons ensuite des différents modèles existants. Enfin, nous présenterons le problème que nous avons étudié : le VERTEX COVER. Nous donnerons un résultat original dans la section 1.4.2, nous donnerons ensuite un exemple d'application sur de grandes tailles puis nous ferons un tour d'horizon des principaux algorithmes existants pour ce problème.

1.1 Contexte, motivations et problématiques

De nos jours, de plus en plus de domaines d'application tels que la biologie, la météorologie ou la finance produisent de très grandes quantités de données (une loi dérivée de la Loi de Moore stipule même que « la quantité de données double tous les 20 mois »). Ces données sont généralement éparpillées sur plusieurs sites et sont regroupées dans de grandes bases de données réparties appelées *entrepôts de données* [70], afin d'être mises à la disposition d'un grand nombre d'utilisateurs, dans le but d'être exploitées et analysées.

Toute une communauté s'intéresse déjà aux grandes bases de données réparties et aux problèmes qui y sont liés, à savoir l'intégration des données, la gestion des accès concurrents, *etc.* En ce qui nous concerne, nous nous intéressons plutôt à la résolution de problèmes algorithmiques d'optimisation sur de grandes instances de données.

La plupart des algorithmes d'optimisation ont besoin d'explorer, de marquer, de modifier, *etc.* l'instance fournie en entrée avant de produire leurs résultats. Pour cela, l'instance est entièrement chargée dans la mémoire de l'ordinateur qui exécute l'algorithme. Souvent même, des structures de données additionnelles sont nécessaires pour mémoriser des paramètres utilisés tout au long des

calculs ou pour construire la solution qui sera retournée à la fin de l'exécution.

Cependant, avec le développement important de données de grande taille, ce modèle de calcul n'est plus forcément pertinent. En effet, la production de grandes quantités d'informations est souvent le fruit de procédés lourds et coûteux (mesures expérimentales, collectes de valeurs provenant de plusieurs sources, *etc.*). Elles doivent donc être préservées de toute modification afin de ne pas être corrompues. C'est l'une des raisons pour laquelle la plupart des utilisateurs ne disposent que d'un accès en lecture seule sur les entrepôts de données. De plus, leurs moyens sont limités : ils ne possèdent pas toujours une machine capable de charger entièrement les données en mémoire et comme le traitement de grandes instances prend du temps, ils ne peuvent pas allouer en général toutes leurs machines durant une trop longue période.

Notre travail s'inscrit dans ce contexte particulier, qui fait l'objet de nombreuses contraintes.

La notion de grandes instances de données est relative. En effet, si l'on est amené à traiter une quantité d'information « normale » avec très peu de moyens (récupérer et traiter des centaines de Gigas de données avec un smartphone par exemple), les problématiques évoquées ci-dessus n'en demeurent pas moins identiques : la notion de mémoire limitée s'applique de la même façon.

1.2 Un modèle naturel de traitement des grandes instances de données

En tenant compte des considérations établies dans la section précédente, nous avons naturellement proposé le modèle suivant (la figure 1.1 en donne une illustration).

Nous supposons qu'il y a une machine standard, appelée « Unité de traitement », pour accéder aux données distantes et pour exécuter les algorithmes. Les données traitées en entrée des algorithmes sont stockées sur un élément externe (un entrepôt de données par exemple) appelé « Instance ». Comme la taille de la solution calculée peut être importante, nous supposons qu'elle est stockée elle aussi sur un élément externe (un disque dur ou un entrepôt de données par exemple) appelé « Résultat ». Nous énumérons maintenant les contraintes principales de notre modèle.

C_1. Les données fournies en entrées ne doivent pas être modifiées : l'*intégrité* de l'instance doit être préservée.

C_2. L'unité de traitement dispose d'un espace mémoire de « petite » taille (par rapport à la grande taille de l'instance).

C_3. Les éléments de la solution doivent être envoyés « à la volée », c'est-à-dire à chaque fois qu'ils sont calculés.

La contrainte C_2 implique que l'instance ne peut pas être chargée dans la mémoire de l'unité de traitement. Plus précisément, nous supposons qu'un nombre très limité d'éléments sont stockables sur l'unité de traitement.
La contrainte C_3 vient du fait que dans de nombreux cas, la taille de la solution produite est (en ordre de grandeur) aussi importante que la taille de l'instance traitée. Il n'est donc pas possible

de la conserver dans la mémoire de l'unité de traitement. Aussi, utiliser des bouts de solutions intermédiaires pour construire la solution finale peut être complexe et coûteux en temps et en espace mémoire. En effet, cela implique le rechargement à partir de la machine « Résultat » des morceaux appropriés. Afin d'éviter de tels mécanismes, nous adoptons un point de vue radical en proposant des méthodes qui scannent les données et envoient les résultats finaux dès qu'ils sont produits, sans conserver en mémoire des traces des calculs déjà effectués et sans modifier des parties de la solution déjà calculées.

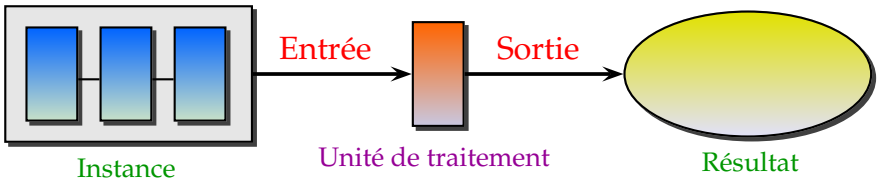


FIG. 1.1 – Vue d'ensemble de notre modèle

Il s'avère que ce modèle est proche de plusieurs modèles existants. Dans la prochaine section, nous allons en dresser un bref récapitulatif.

1.3 D'autres approches liées aux grandes quantités de données

Le modèle que nous avons présenté dans la section précédente présente de nombreux points communs avec des modèles existants tels que le modèle *online*, le modèle *streaming* et le modèle *I/O-efficient*. Nous présenterons dans cette section ces trois modèles, ainsi que l'approche *property testing*.

1.3.1 Le modèle *online*

De nombreux problèmes algorithmiques rencontrés en pratique sont *online* [8]. C'est le cas notamment en réseau, en ordonnancement ou bien encore dans la gestion des ressources d'un système d'exploitation. Dans ces problèmes, l'instance traitée est dévoilée au fur et à mesure durant l'exécution : les algorithmes utilisés pour les résoudre ne disposent pas de la totalité des données et surtout, ils ne connaissent pas le futur, c'est-à-dire qu'ils ne maîtrisent pas la manière dont arrivent ces données. Ils doivent donc construire leurs solutions au fur et à mesure, uniquement avec les informations dont ils disposent, en faisant des choix irrévocables.

Une contrainte importante pour un algorithme online est qu'il doit maintenir à tout moment de son exécution une solution *réalisable*, c'est-à-dire une solution possible pour le problème considéré, en rapport avec les données qu'il a déjà reçues.

Pour évaluer la qualité des solutions produites par un algorithme online, on se base sur le *rapport de compétitivité* (voir [29]). Il s'agit de comparer ses résultats avec ceux d'un algorithme *offline optimal*, c'est-à-dire avec ceux d'un algorithme qui disposerait de la totalité de l'instance tout au long de l'exécution et qui retournerait la meilleure des solutions.

Notre modèle décrit précédemment présente beaucoup de similitudes avec le modèle online. En effet, comme l'unité de traitement n'a pas suffisamment de mémoire pour charger l'instance à traiter en entier, elle la récupère morceau par morceau. Aussi, pour les mêmes raisons, les éléments de la solution sont calculés au fur et à mesure durant l'exécution des algorithmes, qui font de ce fait des choix irrévocables.

Toutefois, dans le modèle online, aucune contrainte n'est énoncée quant à la quantité de mémoire disponible. Bien souvent, l'instance, qui est dévoilée au fur et à mesure, grossit petit à petit : elle est disponible dans sa totalité à la fin de l'exécution, ce qui n'est pas possible dans notre modèle.

1.3.2 Les algorithmes *streaming*

Bien qu'ils aient déjà été étudiés dans les années 1980 [61], les algorithmes streaming ont été formalisés dans un papier de *N. Alon et al.* paru en 1996 [10]. Depuis, de nombreux travaux ont été menés à ce sujet (voir par exemple [37, 59] et plus récemment [75, 130]) et leur champ d'application est vaste [97, 98].

Dans le modèle streaming [22], les données arrivent de manière arbitraire et continue (le terme *data stream* signifie littéralement « flux de données »). Ces données sont accessibles en lecture (elles ne sont pas modifiables) et peuvent être lues une ou plusieurs fois.

L'un des principes fondamentaux de ce modèle (et qui nous intéresse tout particulièrement) est que la quantité de données à traiter est trop conséquente pour être stockée en mémoire. Les possibilités de calcul sont donc restreintes : on doit traiter les informations au fur et à mesure qu'elles arrivent.

Cependant, contrairement au modèle online, les algorithmes ne sont pas tenus de faire des choix irrévocables et de maintenir une solution réalisable à tout moment durant l'exécution (lorsque c'est le cas, on parle alors d'algorithmes online streaming [75]). En effet, dans de nombreux cas, pour la lecture des données, on considère qu'une fenêtre de taille fixe « glisse » sur le flux à traiter : on parle alors de *sliding window* (voir par exemple [60]). On peut voir les données comme un ruban continu et la fenêtre comme une tête de lecture munie d'un tampon d'une certaine taille, cette tête de lecture avançant progressivement dans une direction. Ainsi, lorsqu'elle se déplace, une partie des données contenues dans son tampon reste (voir la figure 1.2). De ce fait, les algorithmes peuvent revenir en arrière dans leurs décisions, mais cette possibilité concerne seulement une infime portion des données.

(a) Lecture des valeurs b, c et d par la fenêtre (b) Etape suivante : lecture des valeurs c, d et e

FIG. 1.2 – Exemple de flux de données lu avec une *sliding window* de taille 3

Pour évaluer les performances d'un algorithme streaming, on s'intéresse particulièrement à deux critères : le nombre de *passes* effectuées, c'est-à-dire le nombre de fois que l'algorithme lit le flux de données en entrée, et la taille de l'espace mémoire disponible. En règle générale, on cherche à minimiser à la fois le nombre de passes effectuées (certains modèles imposent même d'en n'effectuer qu'une seule) et la quantité de mémoire utilisée. Ces deux critères sont liés. Dans [52], les auteurs montrent que plus on diminue la taille de l'espace mémoire disponible, plus le nombre de passes augmente.

Certains chercheurs ont calculé et prouvé des bornes minimales sur la taille de l'espace mémoire nécessaire pour résoudre différents problèmes. Par exemple, *J. Feigenbaum et al.* ont montré dans [59] qu'il était impossible de tester en une seule passe un certain nombre de propriétés dans un graphe avec un espace mémoire de taille inférieure à n, où n représente le nombre de sommets du graphe. Ce résultat (plutôt négatif) a entraîné la mise au point de modèles dérivés plus souples (*T. O'Connel* en fournit un aperçu dans [100]).

1. Le modèle *semi-streaming* [58] : il relâche les restrictions sur les quantités d'espace mémoire disponible. En effet, sa taille est en $\Theta(n \log^k n)$, où k est une constante non nulle (typiquement, pour un graphe à n sommets, on peut stocker l'ensemble de ses sommets et une partie de ses arêtes). Plusieurs travaux ont été menés récemment à ce sujet [6, 56].
2. Le modèle *W-stream* [112] : dans ce modèle, les algorithmes ont la possibilité de modifier le flux de données qu'ils traitent en entrée. Ainsi, lors de chaque passe, ils lisent le flux qui a été modifié lors de la passe précédente. Parmi les travaux récents, on peut citer [51].
3. Le modèle *stream-sort* [5, 112] : il s'agit d'une extension du modèle *W-stream*. En plus de pouvoir modifier les données lues en entrée, les algorithmes peuvent aussi les réorganiser dans le flux. Parmi les travaux portés dans ce modèle, on peut citer [38].

Ces trois modèles étendus sont donc motivés en grande partie par le fait que le modèle streaming classique est trop contraignant. En offrant plus de pouvoir aux algorithmes, on espère ainsi diminuer (suivant les cas) le nombre de passes et/ou la quantité d'espace mémoire utilisé. Cependant, même si les justifications techniques apportées pour valider ces trois modèles sont recevables, dans notre cas, par rapport aux motivations que l'on a évoquées pour énoncer les contraintes fortes de notre modèle, il nous est difficile de dire que ces trois extensions sont bien adaptées au traitement de grandes instances de données.

Le modèle que nous avons décrit dans la section 1.2 présente de nombreux points communs avec le modèle streaming. En effet, en raison de leur quantité importante, l'unité de traitement est contrainte de récupérer les données morceau par morceau. De plus, elle dispose d'un espace mémoire de taille limitée et a un contrôle restreint sur l'instance à traiter puisqu'elle ne peut pas la modifier.

Cependant, dans notre modèle, comme l'unité de traitement ne peut conserver la solution en mémoire, elle doit la construire au fur et à mesure, ce qui implique que les algorithmes utilisés doivent effectuer des choix irrévocables.

1.3.3 L'approche *I/O-efficient*

Formalisé par *A. Aggarwal et J. Vitter* à la fin des années 1980 [4] et étendu dans les années 1990 [122], le modèle I/O-efficient se focalise sur le nombre d'entrées/sorties effectuées par les algorithmes.

Pour être plus précis, dans ce modèle, le processeur (qui exécute les algorithmes) dispose d'une mémoire interne pouvant contenir M éléments et de D disques de grande taille (qui contiennent les données à traiter, au nombre de N). Lors d'une opération d'entrée (resp. de sortie), le processeur charge en mémoire (resp. écrit sur les disques) D blocs de données de taille B chacun. On suppose que $M < N$ et que $1 \leq DB \leq \frac{M}{2}$.

Il est admis que le processeur mette plus de temps à communiquer avec les disques qu'avec la mémoire interne. Les algorithmes I/O-efficient sont donc conçus pour effectuer le moins d'entrées/sorties (E/S) possible. On mesure alors leur la complexité en comptabilisant le nombre d'E/S qu'ils effectuent. Pour cela, on se réfère aux coûts d'opérations élémentaires. Les deux valeurs les plus employées sont :

- $Scan(N)$, qui exprime le coût en E/S pour scanner l'ensemble des données :

$$Scan(N) \in \Theta\left(\frac{N}{DB}\right) \quad .$$

- $Sort(N)$, qui exprime le coût en E/S pour trier l'ensemble des données :

$$Sort(N) \in \Theta\left(\frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B}\right) \quad .$$

  Ce nombre est relativement conséquent, en raison du fait que trier N éléments dans ce modèle est une opération onéreuse.

On retrouve parfois d'autres valeurs :

- $Search(N)$, qui exprime le coût en E/S pour trouver une valeur parmi les N données :

$$Search(N) \in \Theta\left(\log_{DB} N\right) \quad .$$

- $Perm(N)$, qui exprime le coût en E/S pour effectuer une permutation sur l'ensemble des données :

$$Perm(N) \in \Theta\left(\min\left\{N, Sort(N)\right\}\right) \quad .$$

- $Output(Z)$, qui exprime le coût en E/S pour écrire Z éléments sur les disques :

$$Output(Z) \in \Theta\left(\max\left\{1, \frac{Z}{DB}\right\}\right) \quad .$$

Pour plus de détails sur ces valeurs, voir par exemple [121].

De nombreux problèmes de graphe ont fait l'objet d'études dans ce modèle [92], tels que la construction d'arbres couvrants [15, 16] ou le calcul de plus courts chemins [90, 91]. La plupart des algorithmes I/O-efficient proposés ont une complexité qui s'exprime en fonction de $Sort(N)$ (par

exemple, la recherche du diamètre dans un graphe s'effectue en $\mathcal{O}(n \cdot Sort(m))$, avec n le nombre de sommets du graphe et m son nombre d'arêtes).

On peut faire le lien avec le modèle que nous avons décrit dans la section 1.2. En effet, ce que nous avons appelé « Unité de traitement » correspond au processeur, qui dispose d'une mémoire de petite taille (puisque $M < N$), et ce que nous avons appelés « Instance » et « Résultat » correspondent aux disques externes. Toutefois, dans le modèle I/O-efficient, le disque externe (qui est accessible en lecture et en écriture) est aussi utilisé comme extension de la mémoire interne (lorsqu'il n'y a plus assez de place sur celle-ci), ce qui n'est pas le cas dans notre modèle où l'on cherche plutôt à minimiser son utilisation.

Proposé par *J. Abello et al.*, le modèle I/O-efficient fonctionnel [2] n'autorise qu'un accès en lecture aux données. Cependant, les disques sur lesquels elles reposent sont toujours accessibles en écriture : les résultats produits durant l'exécution des algorithmes sont donc disponibles, ce qui n'est pas le cas dans notre modèle, où données et résultats sont stockés sur deux éléments séparés.

1.3.4 *Property testing* et *query complexity*

En 1996, *R. Rubinfeld et M. Sudan* ont défini dans [111] le concept de *property testing*. Leurs travaux, qui portaient sur le test de propriétés algébriques de fonctions, ont eu un rôle important dans le contexte de la vérification de programmes et plus particulièrement dans le contexte des $\mathcal{PCP}$ (voir par exemple [17, 18, 73] pour plus de précisions à ce sujet).

Peu de temps après, *O. Goldreich et al.* [69] ont appliqué ce concept aux graphes. Depuis, ce domaine a fait l'objet de nombreuses études [68, 109].

Un algorithme de property testing est un algorithme qui effectue un petit nombre de requêtes locales sur un objet mathématique pour déterminer s'il possède une propriété globale ou pas (par exemple, tester la connexité d'un graphe en faisant des requêtes pour tester l'existence d'arêtes). Plus précisément, il s'agit d'un *algorithme randomisé* [96] qui résoud correctement un problème de décision avec une probabilité d'au moins $\frac{2}{3}$. Lorsque l'objet traité en entrée ne satisfait pas la propriété testée, on dit qu'il en est ϵ-*éloigné*, avec ϵ un paramètre de distance tel que $0 < \epsilon < 1$.

La complexité des algorithmes de property testing, évaluée en fonction du nombre de requêtes effectuées (et appelée *query complexity*), dépend de plusieurs paramètres tels que la taille de l'instance et le paramètre de distance ϵ. Elle est souvent sous-linéaire en la taille de l'instance, c'est-à-dire que les algorithmes effectuent strictement moins de requêtes qu'il n'y a d'éléments dans l'instance. Ce nombre de requêtes peut même parfois être indépendant de la taille de l'instance traitée. Les algorithmes de property testing sont donc adaptés pour tester des propriétés sur de grandes instances (voir par exemple [67]).

Dans cette approche, on cherche à minimiser le nombre de requêtes effectuées par les algorithmes sur les instances qu'ils traitent. Cependant, aucune contrainte n'est énoncée à propos de la quantité de mémoire disponible.

1.3.5 Synthèse

Les différents modèles que nous avons abordés dans cette section présentent chacun un lien avec le traitement de grandes instances de données.

Dans le modèle online, on insiste sur le fait que l'instance est dévoilée petit à petit, au fur et à mesure. Il faut alors traiter les données « à la volée », c'est-à-dire dès qu'elles arrivent, en faisant des choix irrévocables.

Dans le modèle streaming, les données arrivent aussi de façon arbitraire, morceau par morceau. De plus, la mémoire disponible pour traiter ces données est limitée. Il faut alors restreindre son utilisation et minimiser le nombre de passes effectuées.

Dans le modèle I/O-efficient, les données (à lire et à écrire) sont stockées sur des éléments externes de grande taille. L'accès à ces éléments est coûteux. De plus, la mémoire interne dont on dispose est restreinte. On cherche alors à minimiser le nombre d'accès aux éléments externes.

Dans l'approche property testing, on veut minimiser le nombre de requêtes (locales) effectuées sur l'instance lue en entrée.

De façon générale, le modèle que nous avons proposé dans la section 1.2 reprend une partie des propriétés rencontrées dans ces modèles[1]. En effet, comme dans les modèles streaming et I/O-efficient, la quantité de mémoire disponible est restreinte. De plus, comme la communication avec les éléménts externes est coûteuse, il faut faire des choix irrévocables afin d'envoyer les résultats « à la volée ». (Nous verrons aussi par la suite dans les prochains chapitres que, comme dans les modèles online et streaming, on ne maîtrise pas la façon dont arrivent les données.)

Nous avons choisi d'étudier ce modèle de traitement contraignant sur un problème classique en optimisation combinatoire, le problème du VERTEX COVER, que nous présentons dans la prochaine section.

1.4 Le problème du Vertex Cover

Le problème du VERTEX COVER est probablement l'un des problèmes de combinatoire les plus étudiés en informatique depuis les résultats de *D. Kőnig* parus dans les années 1930 [81]. Il s'agit de plus d'un des problèmes **NP**-complets les plus connus [64]. En effet, en 1972, un an après que *S. Cook* et *L. Levin* aient montré que le problème SAT était **NP**-complet [41, 87], *R. Karp* a prouvé la **NP**-complétude de 21 problèmes, dont celui du VERTEX COVER [80]. Il s'agit d'un problème de graphe qui est employé dans la résolution de nombreux problèmes émanant de différents domaines.

L'intérêt pour ce problème ne faiblit pas depuis les années 1970. En effet, de nombreux travaux ont été menés récemment à ce sujet [1, 19, 26, 31, 72, 83, 117, 120] et ce sous différents aspects : étude sur des familles de graphes particulières, étude dans le cadre de l'algorithmique répartie, *etc.*

On peut plus particulièrement citer les travaux de *F. Delbot* qui, durant sa thèse, a montré que le rapport d'approximation en pire cas ne suffisait pas toujours pour comparer les performances

[1] Toutefois, les motivations évoquées et les justifications fournies ne sont pas toujours les mêmes.

de deux algorithmes [46]. Nous avons repris une partie de ces travaux, notamment au niveau des heuristiques présentées et étudiées, et nous les avons adaptés au contexte du traitement de grandes instances de données.

1.4.1 Présentation

Le problème du VERTEX COVER est un problème défini sur des graphes simples, non orientés et non pondérés. Il en existe de nombreuses variantes, telles que le WEIGHTED VERTEX COVER, le k-PATH VERTEX COVER [30], le CONNECTED VERTEX COVER [57, 133] et bien d'autres encore [24, 28, 95].

Nous donnons dans le paragraphe qui suit des notations qui seront utilisées tout au long de ce document.

Notations générales. Soit $G = (V, E)$ un graphe quelconque, avec V l'ensemble de ses sommets et E l'ensemble de ses arêtes, tel que $E = \{uv \mid u \in V \wedge v \in V\}$. Les graphes $G = (V, E)$ considérés tout au long de ce document sont simples, non orientés et non pondérés. On désigne par n le nombre de sommets ($n = |V|$) et par m le nombre d'arêtes ($m = |E|$). Pour tout sommet $u \in V$, on désigne par $N(u)$ l'ensemble des *voisins* de u (l'ensemble des sommets partageant une arête avec u), $d(u) = |N(u)|$ le *degré* de u (son nombre de voisins) et Δ (resp. δ) le *degré maximum* (resp. *minimum*) des sommets de G.

Définition 1 (Couverture de sommets[2])**.** *Soit $G = (V, E)$ un graphe quelconque. Une* couverture de sommets *C de G est un sous-ensemble de sommets tel que chaque arête possède (ou* est couverte par*) au moins une extrémité dans C. Formellement, on a $C \subseteq V$ et $\forall uv \in E$, on a $u \in C$ ou $v \in C$ (ou les deux).*

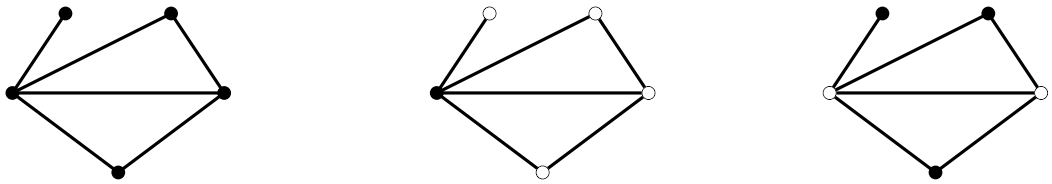
Le problème du VERTEX COVER (dans sa version *problème d'optimisation*) consiste à trouver une couverture de taille minimum. Il s'agit donc d'un problème de minimisation.
Nous désignerons par $OPT(G)$ la taille d'une couverture minimale pour un graphe G.

La figure 1.3 fournit un exemple pour ce problème. La figure 1.3(a) représente un graphe à cinq sommets et six arêtes. Les sommets en blanc de la figure 1.3(b) forment une couverture de sommets du graphe précédent. En effet, chaque arête possède au moins une extrémité dans cette solution. En revanche, bien qu'elle soit *minimale pour l'inclusion*[3], elle n'est pas de taille minimum. La figure 1.3(c) fournit une couverture optimale pour le graphe de la figure 1.3(a).

De nombreux problèmes issus de domaines divers et variés nécessitent la recherche d'une couverture de taille minimum (le plus souvent lors d'une étape intermédiaire) [105]. C'est le cas par exemple en biologie [35, 110, 116], en physique [55, 76] ou bien dans le domaine des réseaux [86, 132].

[2]Le terme *vertex cover* est traduit de différentes façons dans la littérature française (on emploie même parfois le terme *transversal*). Nous utiliserons tout au long de ce document les termes « couverture de sommets » puis (plus simplement) « couverture ».

[3]Une couverture est dite *minimale pour l'inclusion* si et seulement si le fait de lui enlever un sommet lui fait perdre sa qualité de couverture. Une couverture optimale de taille minimum est toujours minimale pour l'inclusion.



(a) Graphe à 5 sommets et 6 arêtes (b) Couverture de sommets (c) Couverture optimale

FIG. 1.3 – Exemple de graphe et de couvertures de sommets

Une des applications « grand public » de ce problème consiste à surveiller l'intégralité des rues d'une ville (ou bien les rayons d'un entrepôt ou d'un magasin) en plaçant un minimum de caméras de vidéo-surveillance aux intersections.

L'appartenance du VERTEX COVER à la classe des problèmes **NP**-complets implique qu'il n'existe pas d'algorithme exact pour le résoudre en temps polynomial (à moins que $\mathbf{P} = \mathbf{NP}$). Il reste même **NP**-complet sur bon nombre de classes de graphes particulières comme les graphes de degré borné [65].

Il existe différentes façons de s'attaquer à un problème d'optimisation. On peut utiliser des algorithmes exacts qui ont une complexité exponentielle ou bien des algorithmes approchés qui ont une complexité polynomiale.

Bien que le problème du VERTEX COVER appartienne à la classe **FPT** (Fixed Parameter Tractable) [54, 62, 99], nous nous sommes tournés vers des méthodes de résolution approchées en temps polynomial. En effet, le fait que ce problème soit dans la classe **FPT** implique qu'il peut être résolu de manière exacte en un temps qui est exponentiel, non pas en la taille de l'instance n, mais en la taille de la couverture optimale k. Cette complexité paramétrique a suscité beaucoup d'intérêt et plusieurs algorithmes exacts ont ainsi été proposés [39, 107], le meilleur algorithme connu à ce jour s'exécutant en $\mathcal{O}(1.2738^k \cdot n^{\mathcal{O}(1)})$ [40]. Ces algorithmes sont efficaces lorsque la taille de la couverture optimale est petite [3, 108]. Mais en pratique, ils ne sont plus utilisables dès lors que cette taille est grande, ce qui est malheureusement souvent le cas sur de gros graphes. Il est donc préférable d'utiliser des algorithmes approchés (d'autant plus que dans la section 1.4.2, nous fournirons une raison supplémentaire de préférer les algorithmes approchés aux algorithmes exacts).

Le problème du VERTEX COVER appartient à la classe **APX** (pour *approximable*), ce qui signifie qu'il admet un algorithme polynomial dont le rapport d'approximation est borné par une constante. A ce jour, le meilleur rapport d'approximation constant connu pour ce problème est 2. Il est atteint avec l'heuristique proposée par *F. Gavril* et qui est basée sur les couplages maximaux (voir [64]). Depuis, de nombreux travaux ont été menés pour améliorer ce facteur. En 1985, *B. Monien et E. Speckenmeyer* [94] et *R. Bar-Yehuda et S. Even* [23] ont proposé des algorithmes avec un rapport d'approximation de $2 - \frac{\ln \ln n}{\ln n}$. Ce ratio a été réduit récemment par *G. Karakostas* à $2 - \Theta\left(\frac{1}{\sqrt{\log n}}\right)$ [79]. Toutefois, aucun facteur constant inférieur à 2 n'a été trouvé pour ce problème.

Ainsi, en 2005, *I. Dinur et S. Safra* ont prouvé que ce problème n'était pas approximable par un facteur inférieur à $10\sqrt{5} - 21 \approx 1.3606$ [53]. Il a aussi été prouvé que ce facteur tendait vers 2 si la *conjecture des jeux uniques* était vraie [82, 104].

1.4.2 Borne minimale sur la quantité de mémoire nécessaire pour construire une couverture optimale

Comme nous avons pu le voir précédemment, dans le modèle streaming, la quantité de mémoire utilisée est un facteur important (il l'est d'autant plus lorsque l'on traite de grandes quantités de données avec des moyens limités). Dans ce qui va suivre, nous allons présenter un résultat général (plutôt négatif) à propos de la quantité minimale de mémoire nécessaire pour calculer en une seule passe une solution optimale pour le problème du VERTEX COVER.

Dans le modèle streaming auquel on s'intéresse ici, les graphes arrivent sur la machine de traitement arête par arête, chaque arête étant composée des identifiants des deux sommets qui la constituent. Le nombre de sommets et le nombre d'arêtes du graphe ne sont pas connus à l'avance.

Le résultat présenté dans cette sous-section est une adaptation des résultats de *M. Zelke* relatifs au problème MIN CUT (voir les pages 49 à 57 de [129] et [130]). Dans ses travaux, il a montré qu'un algorithme streaming fonctionnant en une seule passe nécessitait $\Omega(n^2)$ bits mémoire pour construire une coupe de taille minimum. Pour y parvenir, il a fourni deux preuves distinctes, l'une se basant sur la théorie de la *communication complexity* [84, 127].

Dans ce qui va suivre, nous allons montrer que tout algorithme streaming optimal pour le problème du VERTEX COVER et qui fonctionne en une seule passe nécessite $\Omega(n^2)$ bits d'espace mémoire. Nous donnerons tout d'abord une preuve basée sur la théorie de la *communication complexity* puis nous donnerons une preuve alternative pour ce résultat, formalisé par le théorème 1 qui suit.

Théorème 1. *Soit $G = (V, E)$ un graphe quelconque à n sommets. Soit A^* un algorithme streaming optimal déterministe pour le problème du* VERTEX COVER *et qui fonctionne en une seule passe. La taille de l'espace mémoire nécessaire à l'algorithme A^* pour calculer une couverture optimale de G est en $\Omega(n^2)$ bits.*

Preuve basée sur la théorie de la *communication complexity*

Cette preuve se base tout particulièrement sur le problème (simple) du BIT-VECTOR PROBING [77], qui s'énonce de la manière suivante. Alice possède un vecteur de bits x de taille l, Bob possède un indice i compris entre 1 et l. L'objectif de Bob est de connaître la valeur du bit x_i, sachant que les échanges d'informations ne peuvent aller que d'Alice vers Bob.

Il n'y a pas de méthode plus efficace que celle qui consiste à envoyer le vecteur x en entier à Bob (cela nécessite l bits de communication). Plus précisément, il a été montré dans [84] que n'importe quel protocole qui permet à Bob de récupérer avec une probabilité au moins égale à $\frac{1+\epsilon}{2}$ la valeur de x_i requiert au moins ϵl bits de communication.

L'idée générale de la preuve qui va suivre est de supposer au départ qu'un espace mémoire strictement plus petit que n^2 est suffisant pour construire une solution, afin de montrer au final une contradiction.

Démonstration. Soit A^* un algorithme streaming optimal déterministe pour le problème du VERTEX COVER, qui fonctionne en une seule passe et qui utilise un espace mémoire en $o(n^2)$ bits, c'est-à-dire, en terme d'ordres de grandeur asymptotiques[4], strictement moins que $\Omega(n^2)$. Nous allons utiliser A^* pour construire un protocole dont le coût de communication est en $o(n^2)$ bits pour résoudre le problème du BIT-VECTOR PROBING.

Alice possède un vecteur x de taille $\frac{n(n-1)}{2}$ bits. Elle interprête ce vecteur comme étant la demi-matrice d'adjacence d'un graphe $G = (V, E)$ à n sommets. Elle insère en entrée de A^* les arêtes de G puis elle envoie la configuration mémoire de A^* à Bob. Comme la configuration mémoire de A^* est en $o(n^2)$ bits (d'après l'hypothèse de départ), on a bien un total de $o(n^2)$ bits qui circulent d'Alice vers Bob.

Bob considère son indice i, compris entre 1 et $\frac{n(n-1)}{2}$, comme étant une arête ab dont il veut déterminer l'existence dans le graphe G. Il ne dispose que de la configuration mémoire de A^* (il ne possède pas la matrice d'adjacence du graphe G). Pour déterminer si ab appartient à E, il procède de la manière suivante. Il poursuit l'exécution de A^* en insérant des arêtes supplémentaires (c'est possible, puisque A^* est un algorithme streaming)), étendant G en $G^+ = (V^+, E^+)$, avec $V \subset V^+$ et $E \subset E^+$. Pour chaque sommet $u_i \in V \setminus \{a, b\}$, il crée deux nouveaux sommets v_i et w_i et les connecte à u_i. Il rajoute donc $2n - 4$ arêtes en tout à G pour obtenir G^+ (voir l'exemple de la figure 1.4).

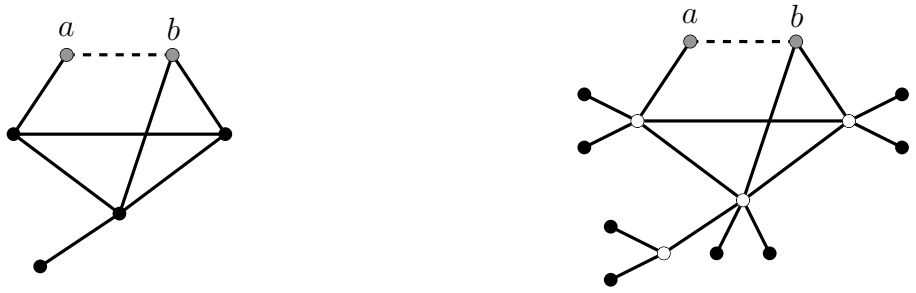
Maintenant, Bob exécute A^* sur G^+ : il obtient une couverture optimale C^* de G^+. Comme chaque sommet $u_i \in V \setminus \{a, b\}$ possède au moins deux voisins distincts de degré 1 dans $G^+ - G$, on a $(V \setminus \{a, b\}) \subseteq C^*$. Si $ab \notin E$, alors a et b ne sont pas dans C^*. En effet, leurs arêtes adjacentes sont déjà couvertes par leurs voisins, qui font tous partie de V. Par contre, si $ab \in E$, alors C^* contient soit a, soit b (voir la figure 1.4).

Au final, nous montrons qu'un algorithme streaming optimal pour le problème du VERTEX COVER, qui fonctionne en une seule passe et qui utilise un espace mémoire en $o(n^2)$ bits, peut être utilisé pour concevoir un protocole de communication transmettant $o(n^2)$ bits et destiné à résoudre le problème du BIT-VECTOR PROBING sur un vecteur de taille $\Theta(n^2)$ bits. Cela contredit la borne minimale de $\Omega(n^2)$ bits transmis dans le pire des cas pour ce problème. □

Preuve alternative pour le théorème 1

Ce résultat peut être prouvé d'une autre manière, sans utiliser la théorie de la *communication complexity*. Auparavant, nous avons besoin d'introduire quelques éléments.

[4]Une définition admise de $o(n)$ est la suivante : $o(f(n)) = \mathcal{O}(f(n)) \setminus \Theta(f(n))$, avec $f : \mathbb{N} \to \mathbb{R}^+$ une fonction.



(a) Graphe G dans lequel Bob cherche à savoir si l'arête ab existe

(b) Graphe G^+ pour lequel la couverture optimale est constituée de tous les sommets en blanc, plus éventuellement a ou b (si l'arête ab existe)

FIG. 1.4 – Graphe dans lequel on veut déterminer l'existence d'une arête

Définition 2 (Configuration mémoire)**.** *Soit* A *un algorithme streaming qui prend un graphe* G *quelconque en entrée. On désigne par* $\mathfrak{M}_{\mathsf{A}}(G)$ *la* configuration mémoire *de l'algorithme* A *après la lecture du graphe* G*.*

Cette preuve alternative se base sur le lemme qui suit, adapté du lemme 12 présenté dans [129].

Lemme 1. *Soit* $\mathcal{G}$ *une famille de* $2^{\Omega(n^2)}$ *graphes construits à partir des mêmes* n *sommets. Pour tout algorithme* A *streaming qui fonctionne en une seule passe et qui utilise* $o(n^2)$ *bits mémoire, il existe deux graphes distincts* $G_1, G_2 \in \mathcal{G}$ *tels que* $\mathfrak{M}_{\mathsf{A}}(G_1) = \mathfrak{M}_{\mathsf{A}}(G_2)$*.*

Démonstration. A tout graphe appartenant à $\mathcal{G}$ (qui contient $2^{\Omega(n^2)}$ graphes différents) correspond une configuration mémoire pour l'algorithme A. Sachant que A dispose d'un espace mémoire en $o(n^2)$ bits, il existe $2^{o(n^2)}$ configurations mémoires différentes, ce qui est strictement inférieur à $2^{\Omega(n^2)}$. Il y a donc forcément deux graphes distincts $G_1, G_2 \in \mathcal{G}$ tels que $\mathfrak{M}_{\mathsf{A}}(G_1) = \mathfrak{M}_{\mathsf{A}}(G_2)$. □

Nous pouvons maintenant donner la preuve pour le théorème 1.

Démonstration. Soit A^* un algorithme streaming optimal déterministe pour le problème du VERTEX COVER, qui fonctionne en une seule passe et qui utilise un espace mémoire de $o(n^2)$ bits. Dans ce qui va suivre, nous allons construire deux graphes distincts G_1^+ et G_2^+ tels que $\mathfrak{M}_{\mathsf{A}^*}(G_1^+) = \mathfrak{M}_{\mathsf{A}^*}(G_2^+)$ mais tels que $OPT(G_1^+) \neq OPT(G_2^+)$.

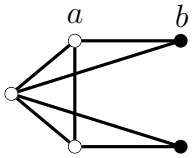
Soit $V = \{u_1, u_2, \ldots, u_n\}$ un ensemble de n sommets, avec $n \mod 3 = 0$. On définit $S \subset V$ par $S = \{u_1, u_2, \ldots, u_{\frac{n}{3}}\}$. Clairement, il y a $2^{\Omega(n^2)}$ graphes distincts constructibles à partir de S. Donc, d'après le lemme 1, il existe deux graphes distincts G_1 et G_2 tels que $\mathfrak{M}_{\mathsf{A}^*}(G_1) = \mathfrak{M}_{\mathsf{A}^*}(G_2)$. Considérons ces deux graphes et supposons que $OPT(G_1) = OPT(G_2)$ (un exemple est donné à la figure 1.5).

Par la suite, pour obtenir G_1^+ et G_2^+, nous allons appliquer exactement les mêmes opérations à G_1 et à G_2, de manière à ce que l'algorithme A^* ne puisse pas les distinguer.

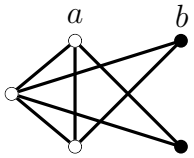
Tout d'abord, on ajoute les $\frac{2n}{3}$ sommets isolés de $V \setminus S$ à G_1 et à G_2, puis on les connecte aux sommets de S de la façon suivante. Puisque $G_1 \neq G_2$, on peut trouver une arête ab appartenant à G_1 mais pas à G_2. Pour tout sommet $u_i \in V \setminus \{a, b\}$, on ajoute une arête $u_i u_{\frac{n}{3}+i}$ et une arête $u_i u_{\frac{2n}{3}+i}$ de la même façon à G_1 et à G_2. On obtient donc deux graphes distincts G_1^+ et G_2^+ où l'arête ab appartient à G_1^+ mais pas à G_2^+.

Comme nous avons effectué exactement les mêmes opérations sur G_1 et sur G_2 et comme $\mathfrak{M}_{\mathsf{A}^*}(G_1) = \mathfrak{M}_{\mathsf{A}^*}(G_2)$, automatiquement, on a $\mathfrak{M}_{\mathsf{A}^*}(G_1^+) = \mathfrak{M}_{\mathsf{A}^*}(G_2^+)$. Reste à savoir si $OPT(G_1^+) = OPT(G_2^+)$.

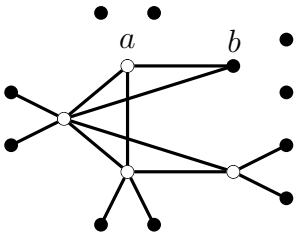
Tels que nous avons construit G_1^+ et G_2^+, les sommets de $V \setminus S$ sont de degré 1 ou 0. Plus précisément, chaque sommet de $S \setminus \{a, b\}$ (il y en a exactement $\frac{n}{3} - 2$) est connecté à exactement deux sommets de degré 1 (il y en a donc $\frac{2n}{3} - 4$ en tout, auxquels s'ajoutent quatre sommets isolés). De ce fait, la couverture optimale pour G_2^+ (dans lequel l'arête ab n'est pas présente) est exactement $S \setminus \{a, b\}$. Donc, $OPT(G_2^+) = \frac{n}{3} - 2$. On ne peut pas avoir la même couverture optimale pour G_1^+. En effet, il faut couvrir l'arête ab qu'il possède et on ne peut pas retirer un sommet de $S \setminus \{a, b\}$, sous peine de rendre au moins deux arêtes non couvertes (voir la figure 1.5). Il faut donc prendre a ou b (en plus des sommets de $S \setminus \{a, b\}$). On a donc $OPT(G_1^+) = OPT(G_2^+) + 1$.



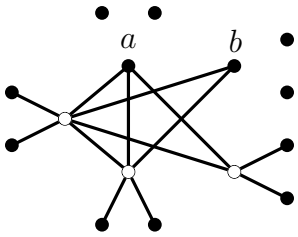
(a) Graphe G_1 tel que $OPT(G_1) = OPT(G_2)$



(b) Graphe G_2 tel que $OPT(G_2) = OPT(G_1)$



(c) Graphe G_1^+ n'ayant pas la même couverture optimale que G_2^+



(d) Graphe G_2^+ n'ayant pas la même couverture optimale que G_1^+

FIG. 1.5 – Construction de deux graphes n'ayant pas la même solution optimale à partir de deux graphes distincts ayant la même couverture optimale

Nous avons donc deux graphes distincts G_1^+ et G_2^+ tels que $\mathfrak{M}_{\mathsf{A}^*}(G_1^+) = \mathfrak{M}_{\mathsf{A}^*}(G_2^+)$. Comme l'algorithme A^* dispose exactement des mêmes éléments en mémoire pour ces deux graphes, il retourne la même solution pour les deux, malgré le fait que $OPT(G_1^+) = OPT(G_2^+) + 1$. Il commet donc une erreur pour l'un d'entre eux. □

1.4.3 Un exemple d'application sur de grands graphes

Comme nous avons pu le voir précédemment, le problème du VERTEX COVER intervient dans la résolution de nombreux problèmes issus de différents domaines. Nous avons aussi vu que les quantités de données manipulées dans la plupart des disciplines sont de plus en plus conséquentes. Nous allons donc fournir dans ce qui va suivre un exemple d'application du problème du VERTEX COVER dans notre modèle de traitement de grandes instances.

Nous allons décrire le problème d'assemblage SNP d'haplotypes, qui intervient dans le séquençage d'ADN. Ce problème a une importance fondamentale, notamment en phylogénie[5] et en médecine (voir [115]). Il a fait l'objet de plusieurs études il y a quelques années [85, 89] et plus récemment [63, 114].

Une molécule d'ADN est une suite de symboles issus d'un alphabet de quatre lettres : A (adénine), C (cytosine), G (guanine) et T (thymine), chaque lettre correspondant à une base d'ADN appelée aussi *nucléotide*. Les chromosomes des organismes *diploïdes* consistent en deux séquences d'ADN appelées *haplotypes* (il y en a une pour chaque brin du chromosome). Un *Single-Nucleotide Polymorphism* (*SNP*, que l'on prononce « snip ») est une différence entre deux nucléotides situés à la même position dans les deux haplotypes. (Pour plus de détails sur ces notions, lire le chapitre 4 de [42] ou bien les pages 95 et 96 de [9].)

Durant le processus de séquençage, en raison des limites technologiques, les longues molécules d'ADN sont découpées en *fragments*. La séquence d'origine est alors reconstituée en assemblant ces fragments. Il s'agit de la *phase d'assemblage*. Celle-ci est d'autant plus compliquée lorsque l'on souhaite reconstituer les deux haplotypes pour chaque chromosome d'un organisme diploïde (c'est le cas notamment pour l'être humain : voir [78]). En effet, en raison des SNPs, certaines régions diffèrent naturellement d'un brin à un autre. Malheureusement, les fragments obtenus peuvent aussi contenir des erreurs. Il faut donc reconstituer l'ADN le plus fidèlement possible, en éliminant un maximum d'erreurs et en retirant un minimum de SNPs. Pour ce faire, il existe principalement trois techniques.

1. Retirer un minimum de fragments.
2. Corriger un minimum de valeurs dans les nucléotides.
3. Ecarter un minimum de SNPs.

Chacune de ces méthodes est liée à un problème d'optimisation [85, 89].

Considérons la troisième technique. Soit $\mathcal{S}$ un ensemble de n SNPs. Soit $\mathcal{F}$ un ensemble de m fragments d'ADN. Soit $R : \mathcal{S} \times \mathcal{F} \to \{0, A, B\}$ une relation indiquant si un SNP $s_i \in \mathcal{S}$ n'apparaît pas dans un fragment $f_j \in \mathcal{F}$ (dans ce cas, $R(s_i, f_j) = 0$) ou s'il apparaît, la valeur non nulle de s_i (dans ce cas, $R(s_i, f_j)$ vaut A ou B). Notons au passage que cette relation R (et en particulier les valeurs A et B) est obtenue uniquement à partir d'observations expérimentales (la figure 1.6 en montre un exemple).

[5] La phylogénie est l'étude des parentés entre différents êtres vivants en vue de comprendre leur évolution.

	s_1 s_2 s_3 s_4
f_1	ATCGATG *C* A *T* G
f_2	TCGATG *G* A *T* G *A* AT
f_3	ATG *C* A *G* G *A* ATGCA *A*
f_4	A *G* G *C* ATGCA *T*
f_5	TGCA *T*

(a) Alignement de fragments d'ADN

R	s_1	s_2	s_3	s_4
f_1	A	A		
f_2	B	A	A	
f_3	A	B	A	A
f_4		B	B	B
f_5				B

(b) Matrice SNP obtenue

FIG. 1.6 – Exemple de fragments d'ADN et de SNPs

Deux SNPs s_i et s_j sont en conflit lorsqu'il existe deux fragments f_k et f_l tels que exactement trois des $R(s_i, f_k)$, $R(s_i, f_l)$, $R(s_j, f_k)$, $R(s_j, f_l)$ ont la même valeur non nulle et une possède la valeur non-nulle opposée (par exemple, si $R(s_i, f_k) = R(s_i, f_l) = R(s_j, f_k) = A$ et si $R(s_j, f_l) = B$).

Soit $G = (\mathcal{S}, \mathcal{C})$ un *graphe des conflits SNP*. Dans ce graphe, chaque sommet $s_i \in \mathcal{S}$ représente un SNP et chaque arête $s_i s_j \in \mathcal{C}$ représente un conflit entre deux SNPs distincts s_i et s_j. La figure 1.7 montre un graphe des conflits SNP généré à partir des valeurs observées dans la figure 1.6.

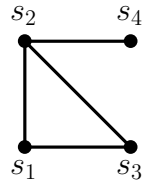


FIG. 1.7 – Exemple de graphe des conflits SNP

Le problème d'assemblage SNP d'haplotypes est donc de maximiser le nombre de SNPs qui ne sont pas en conflit. Une façon de faire est de retirer le plus petit sous-ensemble $\mathcal{S}'$ de $\mathcal{S}$ dans G de telle sorte que le sous-graphe induit $G - \mathcal{S}'$ ne contienne plus d'arête. Cela revient donc à trouver une couverture de taille minimum dans G.

A partir de grandes quantités de mesures expérimentales (obtenues par exemple dans un centre de séquençage d'ADN), on peut générer de très grosses séquences d'ADN et ainsi produire un très gros nombre de SNPs et de fragments. On peut donc créer un très gros graphe des conflits SNP et le stocker (par exemple) sur un entrepôt de données. Ce graphe peut être partagé, via un accès en lecture seule, avec des scientifiques pour effectuer des calculs divers et variés.

Un généticien, qui souhaiterait résoudre des conflits biologiques dans un tel graphe, ne dispose pas nécessairement de machines puissantes pour effectuer les calculs. Ses possibilités sont donc limitées : il ne peut pas copier l'intégralité du graphe dans la mémoire de sa machine (il peut cependant laisser tourner un logiciel pendant plusieurs jours). De ce fait, un procédé de calcul simple doit être implémenté sur sa machine, afin de récupérer le graphe des conflits morceau par morceau (en envoyant des requêtes à l'entrepôt de données), de traiter chacun des morceaux et

d'envoyer le résultat « à la volée » (sur un disque dur externe par exemple).

Il s'agit là d'un exemple parmi tant d'autres puisque la plupart des domaines d'application du VERTEX COVER sont susceptibles de produire à l'heure actuelle de grandes quantités de données.

1.4.4 Tour d'horizon des différents types d'heuristiques existantes

De nombreux algorithmes ont été proposés ces dernières années pour résoudre le problème du VERTEX COVER (voir par exemple la section de [20] consacrée à ce problème). Dans cette sous-section, nous donnons un rapide tour d'horizon des différents types d'heuristiques (les plus classiques) qui existent. Notre objectif est de trouver une heuristique ou un groupe d'heuristiques capables de s'adapter facilement à notre modèle de traitement de grandes instances de données présenté dans la section 1.2.

La solution la plus naturelle consiste à couvrir un maximum d'arêtes en sélectionnant lors de chaque étape un sommet de plus fort degré. C'est ce que fait l'algorithme MDG (pour *Maximum Degree Greedy*).

Algorithme 1 (MDG)**.** Soit $G = (V, E)$ un graphe. Tant que $E \neq \emptyset$, on sélectionne dans G un sommet u de plus fort degré, on l'ajoute à la solution puis on le retire du graphe (les arêtes incidentes à u sont retirées de G et les degrés des sommets restants sont mis à jour).

Il a été prouvé (en adaptant la preuve présente dans [43] pour le problème SET COVER) que cet algorithme retourne des solutions $H(\Delta)$-approchées dans le pire des cas, avec $H(\Delta) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\Delta} \approx \log \Delta$ le *nombre harmonique* de rang Δ . Ce rapport est atteint sur des graphes bipartis particuliers [101] (ces graphes sont appelés *graphes Anti-MDG* dans [46]).

Bien souvent, plusieurs solutions sont possibles : MDG n'est pas déterministe. Lors d'une étape, il peut y avoir plusieurs sommets de plus fort degré (dans ce cas, le choix est arbitraire).

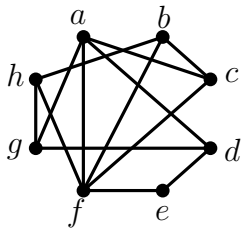
Un exemple d'exécution de l'algorithme MDG est donné par la figure 1.8.

Une autre solution déjà évoquée précédemment consiste à retourner les sommets d'un couplage maximal (au sens de l'inclusion), c'est-à-dire un ensemble d'arêtes deux à deux non adjacentes tel que l'on ne puisse plus ajouter d'autres arêtes. Il s'agit de l'algorithme ED (pour *Edge Deletion*).

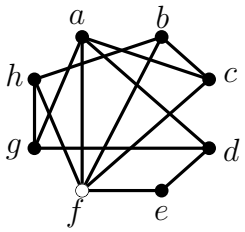
Algorithme 2 (ED)**.** Soit $G = (V, E)$ un graphe. Tant que $E \neq \emptyset$, on sélectionne dans G une arête uv, on ajoute ses extrémités u et v à la solution puis on les retire du graphe (les arêtes incidentes à u et à v sont retirées de G).

Son rapport d'approximation (égal à 2) est atteint sur les étoiles (voir la figure 1.9). Il a aussi été montré qu'il tendait vers $\min\left\{2, \frac{1}{1-\sqrt{1-\epsilon}}\right\}$ pour les graphes ayant un degré moyen d'au moins ϵn et qu'il tendait vers $\min\left\{2, \frac{1}{\epsilon}\right\}$ pour les graphes ayant un degré minimum d'au moins ϵn (voir [32]).

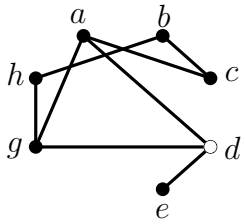
Une autre solution, inspirée par MDG, se base sur le fait que choisir un sommet de degré minimum est un choix peu judicieux (par exemple, dans un arbre, sélectionner une feuille est un



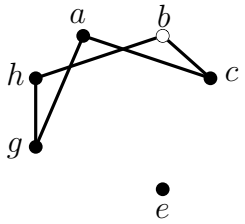
(a) Graphe sur lequel on exécute MDG



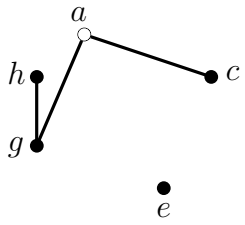
(b) Première étape : on sélectionne le sommet f



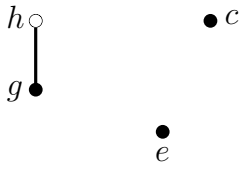
(c) Deuxième étape : on choisit le sommet d



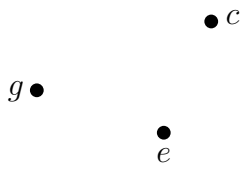
(d) Troisième étape : on prend le sommet b



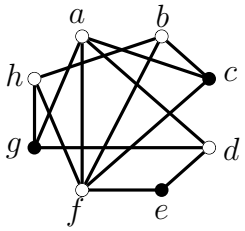
(e) Quatrième étape : on prend le sommet a



(f) Cinquième étape : on choisit le sommet h



(g) Sixième étape : il n'y a plus rien à faire (le graphe ne contient plus d'arête)



(h) Solution obtenue à la fin de l'exécution

FIG. 1.8 – Exécution de l'algorithme MDG sur un graphe à 8 sommets

mauvais choix : en sélectionnant son voisin, on couvre l'arête qui les relie et potentiellement d'autres arêtes). L'algorithme GIC (pour *Greedy Independent Cover*) se base sur ce principe (un exemple

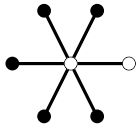
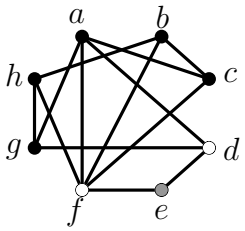


FIG. 1.9 – Solution retournée par l'algorithme ED sur une étoile S_7

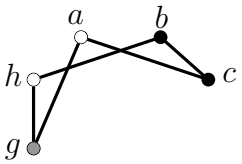
d'exécution est illustré par la figure 1.10).

Algorithme 3 (GIC)**.** Soit $G = (V, E)$ un graphe. Tant que $E \neq \emptyset$, on sélectionne dans G un sommet u de plus faible degré, on ajoute son voisinage $N(u)$ à la solution puis on les retire du graphe (les arêtes incidentes à $N(u) \cup \{u\}$ sont retirées de G et les degrés des sommets restants sont mis à jour).

GIC est au moins $\frac{\sqrt{\Delta}}{2}$-approché (ce rapport est atteint sur les graphes présentés dans [21]). Il est cependant optimal sur les arbres (nous y reviendrons dans le chapitre 2).

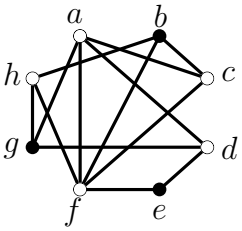


(a) Première étape : on considère le sommet e et on met ses voisins (en blanc) dans la solution



(b) Deuxième étape : on procède de la même façon avec le sommet g et ses voisins a et h

(c) Troisième étape : il ne reste plus que l'arête bc à couvrir



(d) Solution obtenue à la fin de l'exécution

FIG. 1.10 – Exécution de l'algorithme GIC sur le graphe de la figure 1.8

Les algorithmes MDG, ED et GIC décrits ci-dessus ont une complexité en $\mathcal{O}(m)$. De façon générale, ils fonctionnent à chaque étape de la manière suivante : sélectionner un morceau du graphe (un ou plusieurs sommets), en mettre une partie dans la solution puis retirer ces éléments du graphe en le mettant à jour, c'est-à-dire par exemple en recalculant les degrés des sommets

restants. Pour effectuer cela, il faut soit modifier directement le graphe en entrée (ou travailler sur une copie de ce graphe), soit stocker des informations sur les éléments supprimés du graphe. Dans les deux cas, les contraintes C_1 et C_2 de notre modèle décrit dans la section 1.2 ne sont pas satisfaites. De plus, pour MDG (resp. GIC), sélectionner un sommet de plus fort (resp. plus faible) degré demanderait en pratique lors de chaque étape beaucoup d'opérations. Il faudrait en effet calculer à chaque fois les degrés mis à jour de tous les sommets et sélectionner celui qui convient, ce qui donnerait une complexité en $\mathcal{O}(nm)$.

Nous verrons toutefois dans le chapitre 3 qu'en relâchant la contrainte C_2 (en autorisant n bits d'espace mémoire), nous pouvons adapter l'algorithme ED et l'algorithme GIC sur les arbres.

Un autre algorithme connu et proposé en 1982 par *C. Savage* [113] consiste à effectuer un parcours en profondeur et à retourner tous les sommets qui ne sont pas des feuilles. Cet algorithme, communément appelé DFS (pour *Depth First Search*), retourne bien une couverture puisque les feuilles d'un parcours en profondeur ne sont jamais reliées par une arête dans le graphe.

Algorithme 4 (DFS)**.** Soit $G = (V, E)$ un graphe. Soit $T = (V, E_T)$ un arbre construit à partir d'un parcours en profondeur effectué sur G. Soit $F(T)$ l'ensemble des feuilles de T. On met dans la solution l'ensemble des sommets $V \setminus F(T)$.

Il a un rapport d'approximation en pire cas de 2. Ce rapport peut être atteint sur des chemins de taille $2k + 1$. En effet, si la racine à partir de laquelle le parcours en profondeur est effectué est l'une des deux extrémités du chemin, on retourne $2k$ sommets, alors que la taille d'une couverture optimale est k.

Pour calculer un arbre de parcours en profondeur, de la mémoire est nécessaire notamment pour marquer les sommets déjà explorés ou en cours d'exploration. Là aussi, les contraintes C_1 et C_2 ne sont pas respectées.

Une autre catégorie de solutions est basée sur la programmation linéaire et ses extensions. C'est notamment le cas de la solution $2 - \Theta\left(\frac{1}{\sqrt{\log n}}\right)$-approchée de *G. Karakostas*, basée sur de la programmation semi-définie. De manière générale, ces types de méthodes nécessitent le chargement de l'ensemble des contraintes en mémoire, ce qui viole là encore la contrainte C_2.

En s'inspirant de [49] et de [21], *F. Delbot et C. Laforest* ont présenté et étudié dans [47] deux algorithmes de liste, *ListRight* et *ListLeft*, qui ont les propriétés suivantes : ils parcourent tous les deux les sommets du graphe un par un dans un ordre fixé à l'avance (une *liste*), et dès qu'ils traitent un sommet, ils prennent une décision irrévocable à son sujet.

Algorithme 5 (*ListLeft*)**.** Soit $G = (V, E)$ un graphe et $\mathcal{L}$ une *liste* de ses sommets. Soit C la couverture construite (au départ, $C \leftarrow \emptyset$). La liste $\mathcal{L}$ est parcourue *de gauche à droite*. Pour chaque sommet $u \in V$, si u possède un voisin v situé à sa droite dans $\mathcal{L}$ et qui n'est pas dans C, alors on met u dans C.

Algorithme 6 (*ListRight*)**.** Soit $G = (V, E)$ un graphe et $\mathcal{L}$ une *liste* de ses sommets. Soit C la couverture construite (au départ, $C \leftarrow \emptyset$). La liste $\mathcal{L}$ est parcourue *de droite à gauche*. Pour chaque sommet $u \in V$, si u possède un voisin v situé à sa droite dans $\mathcal{L}$ et qui n'est pas dans C, alors on met u dans C.

Il a été montré dans [47] que lorsqu'ils sont exécutés sur un graphe associé à une même liste $\mathcal{L}$, l'algorithme *ListRight* retourne une solution toujours meilleure que l'algorithme *ListLeft*. Il a aussi été montré dans [27] que *ListRight* est 2-approché en moyenne. De plus, des expérimentations menées dans [48] montrent qu'il offre de bonnes performances en moyenne et qu'elles sont même souvent meilleures que celles de l'algorithme ED.

Cependant, durant l'exécution de *ListRight*, il faut conserver en mémoire des informations sur les sommets que l'on a mis dans la solution, ce qui n'est pas le cas pour l'algorithme *ListLeft*. En effet, comme la liste $\mathcal{L}$ est parcourue de gauche à droite, lorsqu'un sommet u est examiné, tous ses voisins situés à sa droite ne sont forcément pas encore dans la couverture. On peut donc redéfinir l'algorithme 5 de la manière suivante (voir l'algorithme 7).

Algorithme 7 (*ListLeft*)**.** Soit $G = (V, E)$ un graphe et $\mathcal{L}$ une *liste* de ses sommets. La liste $\mathcal{L}$ est parcourue *de gauche à droite*. Pour chaque sommet $u \in V$, si u possède un voisin v situé à sa droite dans $\mathcal{L}$, alors on ajoute u dans la solution.

Ainsi, il n'est pas nécessaire de conserver en mémoire des informations sur les sommets que l'on a ajoutés dans la couverture.

Certes, l'algorithme *ListLeft* est plus mauvais que l'algorithme *ListRight*, mais il présente tout de même une propriété importante : il n'a pas besoin de stocker en mémoire des informations durant son exécution. Nous nous sommes donc basés sur cet algorithme pour proposer et étudier dans le chapitre 2 trois heuristiques proches : LL, SLL et ASLL.

Chapitre 2

Etude et analyse de trois algorithmes *memory-efficient* pour le problème du Vertex Cover

Dans ce chapitre, nous nous intéresserons tout particulièrement à trois heuristiques adaptées au traitement de grandes instances de données pour le problème du VERTEX COVER.

Nous avons qualifié ces algorithmes de *memory-efficient*. Cette notion existe déjà dans la littérature, et ce dans différents domaines [36, 44, 126]. Elle est souvent employée de manière relative, afin de comparer les performances mémoires de plusieurs algorithmes entre eux. Par exemple, dans [126], les auteurs présentent un algorithme pour le problème d'alignement structurel d'ARNs dont la complexité mémoire est en $\mathcal{O}(n^3)$. Ils qualifient cet algorithme de memory-efficient puisqu'il améliore la complexité mémoire du précédent algorithme connu qui est en $\mathcal{O}(mn^3)$.

De façon générale, on peut dire qu'un algorithme *memory-efficient* est un algorithme qui n'utilise pratiquement pas de mémoire. Cette définition reste bien évidemment floue puisque, pour chaque problème, on peut quantifier de manière assez libre ce qu'on entend par « pratiquement pas de mémoire ».

Dans notre cas, nous considérons qu'un algorithme *memory-efficient* est un algorithme qui utilise un nombre constant de variables indépendamment de la quantité de données qu'il traite. Les trois algorithmes que nous allons étudier tout au long de ce chapitre n'emploient que deux ou quatre variables pour construire leurs solutions, et ce quelle que soit la taille du graphe lu en entrée (ils ont donc une complexité mémoire en $\mathcal{O}(1)$).

Cependant, la place que prend une variable en mémoire dépend souvent de la taille des données traitées (par exemple, si l'on doit stocker un entier compris entre 1 et n, la variable employée prend $\log_2 n$ bits). Donc, si l'on veut être précis, les algorithmes que nous allons décrire ont une complexité mémoire en $\mathcal{O}(\log n)$ bits ce qui, même en pratique, reste négligeable par rapport à n.

Nous allons donc présenter, étudier et analyser dans ce chapitre trois algorithmes *memory-*

efficient adaptés aux contraintes de notre modèle décrit dans la section 1.2. Après avoir donné leurs descriptions, nous étudierons la qualité des solutions qu'ils retournent, en moyenne et sur des arbres (pour deux d'entre eux), puis nous étudierons leurs complexités.

Une partie des travaux présentés dans ce chapitre ont été exposés dans [13] et publiés dans [14].

2.1 Présentation des algorithmes LL, SLL et ASLL

Nous nous sommes basés sur l'algorithme *ListLeft* présenté et décrit dans la section 1.4 pour définir les trois heuristiques étudiées dans ce chapitre, qui sont par ailleurs librement inspirées de l'algorithme décrit et étudié par *D. Avis et T. Imamura* dans [21].

Avant de donner leurs descriptions, nous devons au préalable définir quelques notions simples qui seront régulièrement utilisées par la suite.

Dans les applications de la vie de tous les jours, les sommets d'un graphe représentent des choses concrètes : des stations de métro, des individus, des tâches à ordonner, *etc.* Bien souvent, chaque sommet d'un graphe possède un identifiant (un *label*) unique. Par exemple, il n'y a pas deux stations du métro parisien qui portent le même nom. Ces différents labels peuvent être ordonnés (par ordre lexicographique par exemple) et donc comparés. Nous le formalisons de la manière suivante.

Définition 3 (Graphe étiqueté)**.** *Un graphe $G = (V, L, E)$ est dit* étiqueté *si ses n sommets sont étiquetés par une fonction $L \in \mathbb{L}(G)$ donnant, pour chaque sommet $u \in V$, un* label *unique $L(u) \in \{1, \ldots, n\}$. $\mathbb{L}(G)$ désigne l'ensemble des $n!$ étiquetages possibles sur le graphe $G = (V, E)$.*

Un exemple de graphe étiqueté est donné à la figure 2.1.

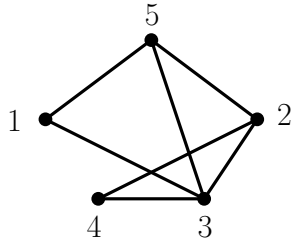


Fig. 2.1 – Exemple de graphe étiqueté à 5 sommets

Définition 4 (Voisin gauche et voisin droit)**.** *Soit $G = (V, L, E)$ un* graphe étiqueté*. Soit $uv \in E$ une arête quelconque de G. Le sommet v est* voisin droit *(resp.* voisin gauche*) du sommet u si et seulement si $L(v) > L(u)$ (resp. $L(v) < L(u)$).*

Nous pouvons maintenant donner une description simple[1] des trois algorithmes LL, SLL (pour *Sorted-LL*) et ASLL (pour *Anti Sorted-LL*) qui se basent sur ces notions. Nous montrerons au passage qu'ils peuvent être facilement implémentés dans notre modèle.

[1]En raison de leur simplicité, nous avons choisi de les présenter sous forme de propriétés ensemblistes. Nous donnerons plus de détails à propos de leur implémentation par la suite.

Algorithme 8 (LL)**.** Soit $G = (V, L, E)$ un *graphe étiqueté*. Pour chaque sommet $u \in V$, u est ajouté à la couverture si et seulement si il possède au moins un *voisin droit*.

L'algorithme SLL est une variante de l'algorithme LL, qui compare d'abord les degrés des sommets avant de comparer leurs labels.

Algorithme 9 (SLL)**.** Soit $G = (V, L, E)$ un *graphe étiqueté*. Pour chaque sommet $u \in V$, u est ajouté à la couverture si et seulement si $\exists v \in N(u)$ tel que $d(v) < d(u)$ ou si u possède un *voisin droit* de même degré.

L'algorithme ASLL est une variante de l'algorithme SLL qui, en quelque sorte, traite le graphe « à l'envers » (puisque les conditions sont inversées).

Algorithme 10 (ASLL)**.** Soit $G = (V, L, E)$ un *graphe étiqueté*. Pour chaque sommet $u \in V$, u est ajouté à la couverture si et seulement si $\exists v \in N(u)$ tel que $d(v) > d(u)$ ou si u possède un *voisin gauche* de même degré.

On peut facilement voir que ces trois algorithmes retournent toujours une couverture, et ce quel que soit le graphe sur lequel ils sont exécutés. En effet, soit uv une arête quelconque. Elle est couverte car l'une de ses deux extrémités u et v est, vis-à-vis de l'autre, pour LL : voisin droit ; pour SLL (resp. ASLL) : voisin de degré inférieur (resp. supérieur) ou voisin droit (resp. gauche) de même degré.

Reprenons le graphe de la figure 2.1. Sur ce graphe, l'algorithme LL retourne l'ensemble de sommets $\{1, 2, 3\}$, tandis que SLL retourne $\{2, 3, 5\}$ et ASLL retourne $\{1, 2, 4, 5\}$.

Nous allons maintenant donner des précisions concernant le traitement des graphes dans notre modèle qui est décrit à la section 1.2. Nous montrerons ensuite que les algorithmes LL, SLL et ASLL sont adaptés à ce modèle.

Précisions à propos du traitement des graphes dans notre modèle

Reprenons le schéma général de notre modèle fourni par la figure 1.1 page 15.

Nous supposons que le graphe $G = (V, L, E)$ pris en entrée est stocké sur la machine « Instance » sous la forme d'une liste d'adjacence dans laquelle les sommets et leurs voisins sont stockés dans un ordre quelconque (pas nécessairement par ordre croissant ou décroissant des labels). Nous supposons que les degrés des sommets sont disponibles sur cette machine lue en entrée : ils peuvent avoir été calculés et stockés lors de la construction du graphe.

L'unité de traitement (qui exécute les algorithmes LL, SLL et ASLL) envoie des *requêtes* à la machine « Instance » et scanne G sommet par sommet. Pour chaque sommet courant u (son label et son degré, si besoin), elle scanne ses voisins (leurs labels et leurs degrés, si besoin) un par un. Lorsque la machine de traitement décide qu'un sommet u est dans la solution (suivant les conditions énoncées précédemment pour chacun des algorithmes), u est envoyé directement et définitivement dans la couverture (cette dernière est stockée sur l'élément extérieur appelé « Résultat »). Alors,

l'unité de traitement récupère le prochain sommet (puis ses voisins, un par un). Dans le cas contraire, elle doit scanner tous les voisins de u (et, à la fin, passer au sommet suivant, comme dans le cas précédent).

On suppose que la machine « Instance » sur laquelle est stockée le graphe a la capacité d'effectuer toutes ces opérations (retourner les labels et les degrés, passer au voisin suivant, *etc.*) de manière efficace.

Les contraintes C_1, C_2 et C_3 sont satisfaites

Montrons maintenant que les trois heuristiques décrites précédemment sont bien adaptées à notre modèle.

Par rapport aux précisions apportées ci-dessus, regardons si les contraintes du modèle C_1, C_2 et C_3 données à la page 14 sont respectées par les trois algorithmes que nous avons présentés.

1. L'instance traitée n'est pas modifiée : l'unité de traitement n'effectue que des opérations de lecture pour comparer deux sommets u et v.
2. Quelle que soit la taille du graphe G en entrée, à tout moment, l'unité de traitement stocke uniquement deux sommets (leurs labels et éventuellement leurs degrés).
3. Les sommets de la couverture construite par les algorithmes sont mis dans la solution dès que possible, et ce de manière définitive.

Par conséquent, LL, SLL et ASLL satisfont les contraintes de notre modèle. Ils sont donc adaptés pour traiter des grandes instances.

Propriétés remarquables de ces algorithmes

En plus d'être adaptés au modèle de traitement de grandes instances de données présenté lors du chapitre 1, les trois heuristiques présentent des propriétés intéressantes.

Ils peuvent très bien s'adapter à un modèle dans lequel les graphes seraient lus arêtes par arêtes. Plus spécifiquement, l'algorithme LL, qui n'a pas besoin de connaître les degrés des sommets et de leurs voisins, peut facilement s'adapter dans le modèle streaming, où les graphes sont bien souvent traités arête par arête (voir par exemple [75]).

Une autre bonne propriété de ces trois algorithmes est que, pour un graphe étiqueté donné, quel que soit l'ordre de traitement des sommets et de leurs voisins, ils sont déterministes. On peut donc facilement découper la liste d'adjacence en plusieurs morceaux et effectuer les opérations en parallèle. Un exemple d'un tel découpage est donné par la figure 2.2. Dans cet exemple, la liste d'adjacence est répartie sur deux machines, qui en évaluent chacune un morceau de manière indépendante. La première machine, qui évalue les sommets 3 et 1, retourne le sommet 1, tandis que la seconde machine, qui évalue les sommets 4 et 2, retourne le sommet 2. Au final, on a bien comme couverture l'ensemble de sommets $\{1, 2\}$.

De plus, comme l'ont montré *F. Delbot et al.* [46, 47], ces algorithmes peuvent facilement être distribués. Ils produisent même moins de messages/rounds (que ce soit dans le modèle asynchrone

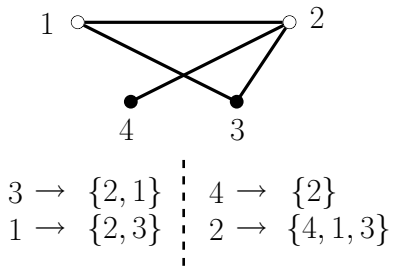


FIG. 2.2 – Exemple d'exécution parallélisée de l'algorithme LL

ou synchrone) que l'algorithme *ListRight* décrit page 33. En effet, dans le modèle asynchrone, l'algorithme LL produit $2m$ messages, tandis que l'algorithme *ListRight* produit $3m$ messages. De plus, dans le modèle synchrone, LL est optimal en nombre de rounds de communication, ce qui n'est pas le cas de *ListRight*.

2.2 Etude de la qualité des solutions retournées

Un critère classique utilisé pour analyser et comparer les performances en terme de qualité de solution des heuristiques est le rapport d'approximation en pire cas.

Les algorithmes LL et ASLL ont un rapport d'approximation d'au moins Δ. En effet, sur les étoiles, où la solution optimale est constituée uniquement du centre, LL peut retourner toutes les feuilles. ASLL a un comportement encore plus mauvais sur ces graphes puisqu'il retourne à chaque fois toutes les feuilles.

L'algorithme SLL a, quant à lui, de meilleurs résultats en pire cas. En effet, *D. Avis et T. Imamura* ont prouvé dans [21] qu'il avait un rapport d'approximation de $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$.

Si l'on s'en tient à ces résultats, on peut dire que l'algorithme SLL est meilleur que les deux autres. Mais les performances en pire cas de tel ou tel algorithme sont parfois éloignées de ses performances en moyenne (voir par exemple les travaux de *F. Delbot* [46]) : un algorithme 2-approché peut très bien avoir un comportement global en moyenne moins bon qu'un algorithme Δ-approché. De plus, le fait d'utiliser ces algorithmes sur de grandes instances de données nous oblige à mener une étude détaillée et approfondie de leurs performances. En effet, des différences théoriques infimes peuvent s'avérer importantes dans ce contexte. Il est donc important de fournir aux utilisateurs le plus d'éléments possible pour les guider au mieux dans leurs choix futurs. Nous avons donc étudié en moyenne les algorithmes LL, SLL et ASLL.

Lorsque l'on exécute ces algorithmes sur un graphe étiqueté, ils ont chacun un comportement déterministe. Cependant, l'étiquetage des sommets d'un graphe est souvent arbitraire et dépend uniquement des domaines d'application. Il y a plusieurs façons d'étiqueter un graphe et pour chaque étiquetage, ces algorithmes peuvent produire des résultats différents.

Dans ce qui va suivre, nous allons étudier, analyser et comparer les qualités moyennes des solutions construites par les algorithmes LL, SLL et ASLL. Pour cela, nous considérerons que les $n!$

étiquetages possibles apparaissent chacun avec une probabilité uniforme de $\frac{1}{n!}$.

2.2.1 Analyse en moyenne

Nous allons donner dans le théorème 2 les formules exactes exprimant les tailles moyennes des solutions construites par LL, SLL et ASLL pour tout graphe G. Auparavant, nous devons introduire quelques notations.

Définition 5. *Soit $G = (V, E)$ un graphe quelconque.*

- *Soit $S = V \setminus \{u \mid \exists v \in N(u),\ d(v) < d(u)\}$ l'ensemble des sommets n'ayant pas de voisin de degré inférieur.*
- *Soit $\overline{S} = V \setminus \{u \mid \exists v \in N(u),\ d(v) > d(u)\}$ l'ensemble des sommets n'ayant pas de voisin de degré supérieur.*
- *Soit $u \in V$ un sommet de G. Soit $\sigma(u) = |\{v \mid uv \in E \wedge d(v) = d(u)\}|$ le nombre de voisins de u ayant le même degré que lui.*

Théorème 2. *Soit $G = (V, E)$ un graphe quelconque. Soit $\mathbb{E}[\mathsf{A}(G)]$ la taille moyenne des solutions construites par l'algorithme* A *pour le graphe G (*A *désigne à la fois* LL, SLL *et* ASLL*). En considérant tous les étiquetages de $\mathbb{L}(G)$ de manière équiprobable, pour* LL*, on a*

$$\mathbb{E}[\mathsf{LL}(G)] \quad = \quad n - \sum_{u \in V} \frac{1}{d(u)+1} \ . \tag{2.1}$$

De la même façon, pour SLL*, on a*

$$\mathbb{E}[\mathsf{SLL}(G)] \quad = \quad n - \sum_{u \in S} \frac{1}{\sigma(u)+1} \ . \tag{2.2}$$

Et pour ASLL*, on a*

$$\mathbb{E}[\mathsf{ASLL}(G)] \quad = \quad n - \sum_{u \in \overline{S}} \frac{1}{\sigma(u)+1} \ . \tag{2.3}$$

Démonstration. On donne les preuves pour LL et pour SLL. La preuve pour ASLL est similaire à celle pour SLL.

Preuve pour LL. Soit $G = (V, L, E)$ un graphe étiqueté. Soit C_{LL} la couverture construite par LL sur G. Considérons un sommet u de G. Ce sommet n'est pas sélectionné par LL si et seulement si il n'a pas de voisin droit, ce qui signifie que tous ses voisins ont des labels plus petits que lui. Comme nous considérons une distribution uniforme des $n!$ étiquetages possibles de $\mathbb{L}(G)$, cet évènement apparaît avec une probabilité de

$$\frac{d(u)!}{(d(u)+1)!} \ .$$

En effet, si l'on trie u et ses $d(u)$ voisins par ordre croissant des labels, il y a $(d(u)+1)!$ permutations possibles (remarquons que pour chaque permutation de u et de ses $d(u)$ voisins, il y a le même

nombre d'étiquetages qui donnent lieu à cette permutation), et le nombre de permutations telles que u soit situé en dernière position est $d(u)!$ (cela revient à placer u à un endroit précis, puis à considérer tous les placements possibles de ses $d(u)$ voisins). Donc,

$$\mathbb{P}\big[u \in C_{\mathsf{LL}}\big] \;=\; 1 - \frac{1}{d(u)+1} \;.$$

Le résultat final s'obtient en effectuant la somme de ces probabilités pour chacun des sommets u de G.

Preuve pour SLL. Soit $G = (V, L, E)$ un graphe étiqueté. Soit C_{SLL} la couverture construite par SLL sur G. Considérons un sommet u de G. Si $u \notin S$, cela signifie qu'il existe un sommet $v \in N(u)$ tel que $d(v) < d(u)$. Donc, pour tous les sommets de $V \setminus S$, on a

$$\mathbb{P}\big[u \in C_{\mathsf{SLL}} \mid u \notin S\big] \;=\; 1 \;.$$

Si $u \in S$, il est sélectionné par SLL si et seulement si il possède un voisin droit de même degré. En appliquant le même raisonnement que pour LL sur tous les sommets de $V \setminus S$, on a

$$\mathbb{P}\big[u \in C_{\mathsf{SLL}} \mid u \in S\big] \;=\; 1 - \frac{1}{\sigma(u)+1} \;.$$

Le résultat s'obtient en effectuant la somme de ces probabilités pour chacun des sommets u de G. □

Le résultat que l'on obtient est similaire aux résultats de *Y. Caro* [33] et de *V. K. Wei* [123] relatifs à la taille d'un ensemble indépendant dans un graphe (voir aussi [11]).

Les formules présentées et décrites dans le théorème 2 peuvent être appliquées de manière analytique en temps polynomial sur des familles de graphes connues.

Afin de tenter d'établir des relations entre ces trois algorithmes, nous avons étudié les tailles moyennes des solutions produites par ces trois algorithmes sur quelques familles de graphes connues.

Exemples d'application sur des familles de graphes

Dans ce qui va suivre, nous présenterons les détails des résultats obtenus sur les étoiles, les grilles et les graphes d'*Avis-Imamura* étendus (une définition de cette famille de graphes sera fournie par la suite). Nous avons choisi ces trois familles de graphes afin de mettre en valeur le fait que, pour chaque algorithme, il existe des graphes pour lesquels il est le meilleur en moyenne.

Sur les étoiles. Soit $S_n = (V, E)$ une étoile à n sommets. Elle est constituée d'un sommet de degré $n-1$, le centre, auquel sont reliés $n-1$ sommets de degré 1, les feuilles. Appliquons respectivement (2.1), (2.2) et (2.3) sur S_n. Pour tout $n > 2$, on obtient

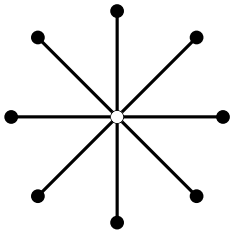
$$\mathbb{E}\big[\mathsf{LL}(S_n)\big] \;=\; n - \frac{n-1}{2} - \frac{1}{n} \;=\; \frac{n}{2} - \frac{1}{n} + \frac{1}{2} \;.$$

Pour SLL, comme on peut le voir sur la figure 2.3(a), l'ensemble S contient toutes les feuilles de S_n. Donc, on a

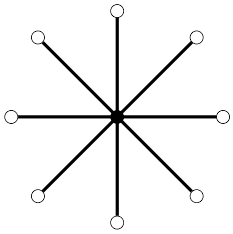
$$\mathbb{E}\big[\mathsf{SLL}(S_n)\big] \;=\; n - n + 1 \;=\; 1 \;.$$

Pour ASLL, comme on peut le voir sur la figure 2.3(b), l'ensemble $\overline{S}$ contient uniquement le centre de S_n. Par conséquent, on a

$$\mathbb{E}\big[\mathsf{ASLL}(S_n)\big] \;=\; n - 1 \;.$$



(a) Mise en valeur des sommets de l'ensemble S, qui sont toujours retournés par SLL



(b) Mise en valeur des sommets de l'ensemble $\overline{S}$, qui sont toujours retournées par ASLL

FIG. 2.3 – Mise en valeur des ensembles S et $\overline{S}$ sur une étoile S_9

On peut facilement voir que

$$\mathbb{E}\big[\mathsf{SLL}(S_n)\big] \;<\; \mathbb{E}\big[\mathsf{LL}(S_n)\big] \;<\; \mathbb{E}\big[\mathsf{ASLL}(S_n)\big] \;.$$

Notons aussi que SLL est optimal sur les étoiles.

Sur les grilles. Soit $GR_{p\times q} = (V, E)$ une grille avec $n = p \times q$ sommets. Elle est constituée de $(p-2)(q-2)$ sommets de degré 4 (les sommets internes), de $2(p+q-4)$ sommets de degré 3 (les sommets du bord) et de 4 sommets de degré 2 (les coins). $\forall p, q > 2$, on obtient

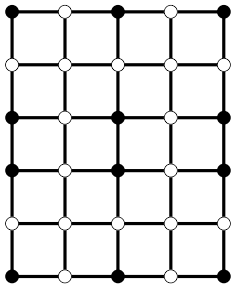
$$\begin{aligned}\mathbb{E}\big[\mathsf{LL}(GR_{p\times q})\big] &= n - \frac{(p-2)(q-2)}{5} - \frac{2(p+q-4)}{4} - \frac{4}{3} \\ &= \frac{4n}{5} - \frac{p+q}{10} - \frac{2}{15} \;.\end{aligned}$$

Pour SLL, comme on peut le voir sur la figure 2.4(a), l'ensemble S contient tous les sommets qui sont voisins du bord et des coins de $GR_{p\times q}$. On a donc

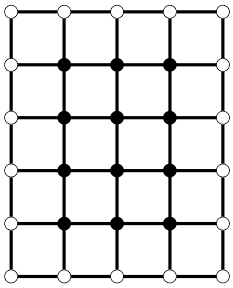
$$\begin{aligned}\mathbb{E}\big[\mathsf{SLL}(GR_{p\times q})\big] &= n - \frac{(p-4)(q-4)}{5} - \frac{2(p+q-8)}{3} - 4 \\ &= \frac{4n}{5} + \frac{2(p+q)}{15} - \frac{28}{15} \;.\end{aligned}$$

Pour ASLL, comme on peut le voir sur la figure 2.4(b), l'ensemble $\overline{S}$ contient tous les sommets du bord et les coins de $GR_{p\times q}$. Par conséquent, on a

$$\begin{aligned}\mathbb{E}\big[\mathsf{ASLL}(GR_{p\times q})\big] &= n - \frac{(p-4)(q-4)}{5} - \frac{2(p+q-8)}{4} - \frac{4}{3} \\ &= \frac{4n}{5} + \frac{3(p+q)}{10} - \frac{8}{15}\ .\end{aligned}$$



(a) Mise en valeur des sommets de l'ensemble S, qui sont toujours retournés par SLL



(b) Mise en valeur des sommets de l'ensemble $\overline{S}$, qui sont toujours retournées par ASLL

FIG. 2.4 – Mise en valeur des ensembles S et $\overline{S}$ sur une grille $GR_{6\times5}$

Donc, on peut voir que LL est meilleur en moyenne que SLL et ASLL sur des grilles.

Définition 6 (Graphes d'*Avis-Imamura* étendus)**.** *Soit $AI_a^+ = (V, E)$ un graphe d'Avis-Imamura étendu de dimension a. Il s'agit d'un graphe biparti spécial avec $n = 2a^2 + a - 1$ sommets, dans lequel :*

- *l'ensemble des sommets est $X_1 \cup X_2 \cup Y_1 \cup Y_2$, avec $X_1 = \{v_1, \ldots, v_{a^2-2}\}$, $Y_1 = \{w_1, \ldots, w_{a^2}\}$, $X_2 = \{z_1, \ldots, z_a\}$ et $Y_2 = \{t\}$;*
- *l'ensemble des arêtes est $v_i w_j$ $\forall i, j$; $z_i w_{a(i-1)+k}$ pour $k = 1, \ldots, a$ et $i = 1, \ldots, a$; et tz_i $\forall i$.*

Un exemple de graphe d'*Avis-Imamura* étendu est donné par la figure 2.5.

Sur les graphes d'*Avis-Imamura* étendus. Soit $AI_a^+ = (V, E)$ un graphe d'*Avis-Imamura* étendu avec $a > 2$. L'ensemble des sommets $V = X_1 \cup Y_1 \cup X_2 \cup Y_2$ se compose de la manière suivante : X_1 contient $a^2 - 2$ sommets de degré a^2, Y_1 contient a^2 sommets de degré $a^2 - 1$, X_2 contient a sommets de degré $a + 1$, et Y_2 contient 1 sommet de degré a. Donc, pour LL, on obtient

$$\begin{aligned}\mathbb{E}\big[\mathsf{LL}(AI_a^+)\big] &= n - \frac{a^2-2}{a^2+1} - \frac{a^2}{a^2-1+1} - \frac{a}{a+2} - \frac{1}{a+1} \\ &= n - 1 - \frac{a^2-2}{a^2+1} - \frac{a}{a+2} - \frac{1}{a+1}\ .\end{aligned}$$

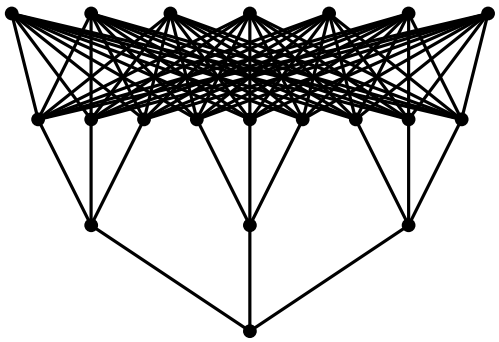


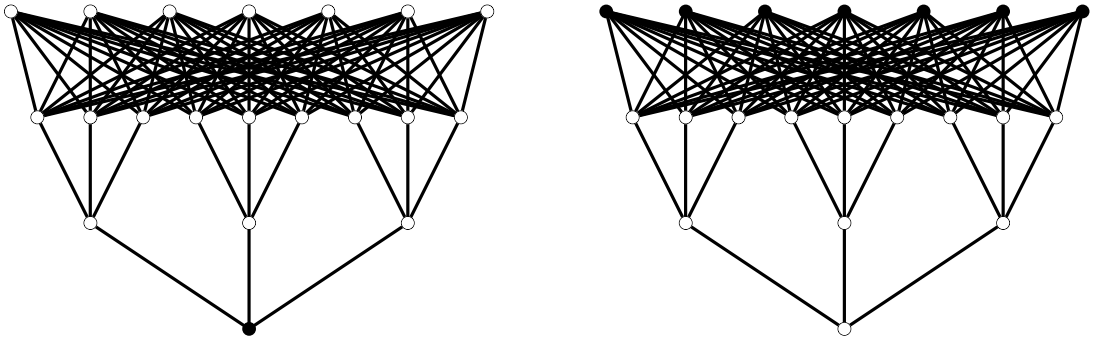
FIG. 2.5 – Graphe d'*Avis-Imamura* étendu de dimension 3

Pour SLL, comme on peut le voir sur la figure 2.6(a), l'ensemble S contient uniquement le sommet t de Y_2. Donc, on a

$$\mathbb{E}\big[\mathsf{SLL}(AI_a^+)\big] \;=\; n-1 \;.$$

Pour ASLL, comme on peut le voir sur la figure 2.6(b), l'ensemble $\overline{S}$ contient les a^2-2 sommets de X_1. Par conséquent, on a

$$\mathbb{E}\big[\mathsf{ASLL}(AI_a^+)\big] \;=\; n-a^2+2 \;.$$



(a) Mise en valeur des sommets de l'ensemble S, qui sont toujours retournés par SLL

(b) Mise en valeur des sommets de l'ensemble $\overline{S}$, qui sont toujours retournées par ASLL

FIG. 2.6 – Mise en valeur des ensembles S et $\overline{S}$ sur un graphe d'*Avis-Imamura* étendu AI_3^+

On peut voir que LL est meilleur en moyenne que SLL. Comparons ASLL et LL.

$$\mathbb{E}\big[\mathsf{LL}(AI_a^+)\big] - \mathbb{E}\big[\mathsf{ASLL}(AI_a^+)\big] \;=\; a^2-3-\frac{a^2-2}{a^2+1}-\frac{a}{a+2}-\frac{1}{a+1} \;>\; 0$$

Lorsque $a>3$, puisque $\frac{a^2-2}{a^2+1}<1$, $\frac{a}{a+2}<1$ et $\frac{1}{a+1}<1$, ce qui implique que $\frac{a^2-2}{a^2+1}+\frac{a}{a+2}+\frac{1}{a+1}<3$. Par conséquent,

$$\mathbb{E}\big[\mathsf{ASLL}(AI_a^+)\big] \;<\; \mathbb{E}\big[\mathsf{LL}(AI_a^+)\big] \;<\; \mathbb{E}\big[\mathsf{SLL}(AI_a^+)\big] \;.$$

Notons que SLL retourne toujours la pire des solutions possible (de taille $n-1$) sur n'importe quel graphe d'*Avis-Imamura* étendu.

Synthèse des résultats obtenus sur d'autres familles de graphes. En plus des trois exemples d'application détaillés ci-dessus, de la même manière, nous avons déterminé de façon analytique les tailles moyennes des solutions retournées par LL, SLL et ASLL sur d'autres familles de graphes. Le tableau 2.1 fournit une synthèse de ces résultats (les détails de ces calculs sont donnés dans [12]).

Algorithmes / **Familles de graphes**	LL	SLL	ASLL
Graphes Δ-réguliers[a]	1er *ex aequo*	1er *ex aequo*	1er *ex aequo*
Chemins	2ème	1er	3ème
Etoiles	2ème	1er	3ème
Graphes bipartis complets	3ème	1er	2ème
Graphes d'*Avis-Imamura*[b]	3ème	1er *ex aequo*	1er *ex aequo*
Graphes d'*Avis-Imamura* modifiés[c]	3ème	2ème	1er
Graphes d'*Avis-Imamura* étendus	2ème	3ème	1er
Grilles	1er	2ème	3ème
Spider graphs[d]	1er	2ème	3ème

TAB. 2.1 – Synthèse des résultats obtenus sur les comparaisons des tailles moyennes des solutions retournées par les trois algorithmes sur certaines familles de graphes

[a] Un graphe Δ-régulier est un graphe dans lequel tous les sommets ont le même degré Δ.

[b] Il s'agit des graphes présentés dans l'article de *D. Avis et T. Imamura* [21].

[c] Un graphe d'*Avis-Imamura* modifié est un graphe d'*Avis-Imamura* étendu auquel on retire la composante Y_2 (constituée du sommet t).

[d] Un *spider graph* $W_{l\times p}$ est le résultat du Produit Cartésien entre un chemin P_p et un cycle C_l.

On peut remarquer que pour chaque algorithme, il existe au moins une famille de graphes pour laquelle il retourne, en moyenne, des solutions meilleures que les deux autres.

Lorsque nous avons commencé à étudier les algorithmes LL, SLL et ASLL, nous pensions que SLL était le meilleur des trois en moyenne. Or, même s'il est plus difficile d'exhiber des familles de graphes pour lesquelles ASLL est le meilleur, SLL peut quand même être très mauvais sur certains graphes, non seulement en moyenne, mais aussi pour toute exécution, comme c'est le cas sur les graphes d'*Avis-Imamura* étendus.

2.2.2 Propriétés spécifiques de LL

Dans cette sous-section, nous allons montrer que pour n'importe quel graphe, l'algorithme LL peut toujours retourner une solution optimale, comme il peut toujours retourner une solution de taille maximum. En d'autres termes, nous allons montrer que LL a toujours une « marge de manœuvre » maximale sur n'importe quel graphe, ce qui n'est pas le cas des deux autres algorithmes.

Théorème 3. *Pour tout graphe G, il existe un étiquetage $L^* \in \mathbb{L}(G)$ tel que l'algorithme* LL *retourne une solution optimale pour le graphe étiqueté $G = (V, L^*, E)$.*

Démonstration. Soit $G = (V, E)$ un graphe quelconque. Soit C^* une couverture optimale de G. On sait que $V \setminus C^*$ est un ensemble indépendant. De plus, chaque sommet $u \in C^*$ possède au moins un voisin dans $V \setminus C^*$ (sinon, u et tous ses voisins seraient dans C^*, donc C^* ne serait pas optimale).

Soit $L^* \in \mathbb{L}(G)$ un étiquetage des sommets de G, tel que les sommets de C^* obtiennent des labels compris entre 1 et $|C^*|$ et ceux de $V \setminus C^*$ des labels compris entre $|C^*| + 1$ et n.

Si l'algorithme LL est exécuté sur le graphe G étiqueté par L^*, il retourne tous les sommets de C^*, puisque chaque sommet u de C^* possède au moins un voisin dans $V \setminus C^*$ ayant un plus grand label. Il ne retourne aucun sommet de $V \setminus C^*$, puisque $V \setminus C^*$ est un ensemble indépendant et que par conséquent, chaque sommet dans cet ensemble possède uniquement des voisins dans C^*, tous ayant par ailleurs un label inférieur à eux. □

Bien évidemment, les algorithmes SLL et ASLL ne peuvent pas faire mieux en terme de borne minimale sur la qualité des solutions retournées puisque, comme nous avons pu le voir précédemment, il existe pour chacun des graphes pour lesquels ils ne sont pas capables de retourner une solution optimale (par exemple les graphes d'*Avis-Imamura* pour SLL et les étoiles pour ASLL).

Théorème 4. *Pour tout graphe G, il existe un étiquetage $L_w \in \mathbb{L}(G)$ tel que l'algorithme* LL *retourne une couverture de taille $n - c$ pour le graphe étiqueté $G = (V, L_w, E)$, avec c le nombre de composantes connexes de G ($c = 1$ si ce graphe G est connexe). Cette borne est atteinte :* LL *ne peut pas construire une couverture de taille supérieure à $n - c$.*

Démonstration. Dans un premier temps, considérons un graphe $G = (V, E)$ avec $c = 1$ composante connexe. Soit $T = (V, E')$ un *arbre couvrant* de G. Soit r un sommet quelconque de T. L'étiquetage $L_w \in \mathbb{L}(G)$ attribue des labels aux sommets de la façon suivante. Le sommet r est étiqueté par n. Les d_1 voisins/enfants de r dans T obtiennent les d_1 labels $n - d_1, \ldots, n - 1$. Les d_2 sommets à distance 2 de r dans T obtiennent les d_2 labels $n - d_1 - d_2, \ldots, n - d_1 - 1$, *etc.* jusqu'à ce que chaque sommet reçoive un label, niveau par niveau. La figure 2.7 fournit un exemple d'un tel étiquetage sur un graphe.

Avec cet étiquetage, puisque T est un arbre couvrant, chaque sommet $u \neq r$ possède au moins un voisin droit : son *père* dans l'arbre T dont la racine est r. Par conséquent, l'exécution de l'algorithme LL sur le graphe G étiqueté par L_w retourne tous les sommets de G, à l'exception de r, qui est le sommet qui possède le label de plus forte valeur. Il s'agit de la taille maximale atteignable, puisque LL ne choisit jamais un sommet étiqueté par le plus grand des labels, car ce dernier ne possède pas de voisin droit.

Si le graphe G considéré n'est pas connexe, on peut appliquer le raisonnement précédent sur chacune de ses c composantes connexes. □

Là aussi, les algorithmes SLL et ASLL ne peuvent pas faire pire. Si l'on considère un graphe connexe de taille finie, l'un comme l'autre ne peuvent pas retourner tous les sommets de ce graphe :

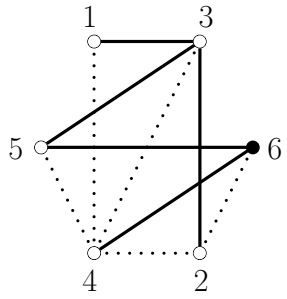


FIG. 2.7 – Graphe étiqueté à l'aide d'un arbre couvrant pour lequel l'algorithme LL retourne $n-1$ sommets (les lignes en pointillés correspondent aux arêtes du graphe qui ne font pas partie de l'arbre couvrant)

ils ne peuvent pas tous avoir de voisin de degré inférieur (ou supérieur) et ils ne peuvent pas tous avoir de voisin droit (ou gauche) de même degré.

2.2.3 Comparaison des algorithmes SLL et ASLL sur les arbres

A la base, notre objectif était de montrer que l'algorithme SLL produit une solution toujours meilleure que l'algorithme ASLL, pour n'importe quel arbre. Nous n'y sommes malheureusement pas parvenus (mais nous continuons à penser qu'il n'existe pas de contre-exemple affirmant le contraire).

Dans cette sous-section, nous montrerons que l'algorithme SLL produit toujours une solution meilleure que l'algorithme ASLL, pour n'importe quel arbre dont au moins la moitié des sommets sont des feuilles et pour n'importe quel chemin.

Dans un premier temps, nous allons montrer certaines propriétés générales liées à l'algorithme OT (que nous allons présenter) et aux feuilles, puis nous allons utiliser ces résultats pour exhiber des propriétés à propos des algorithmes SLL et ASLL sur les arbres, afin de montrer que SLL est meilleur que ASLL.

Bien que **NP**-difficile en général, le problème du VERTEX COVER est polynomial sur les arbres. En effet, il existe un algorithme, que nous avons appelé OT (pour *Optimal Tree*), qui retourne une couverture de taille minimum pour tout arbre. Cet algorithme est une adaptation de l'algorithme GIC que nous avons présenté dans le chapitre 1 (voir l'algorithme 3 page 31).

Algorithme 11 (OT)**.** Soit $T = (V, E)$ un arbre. Tant que $E \neq \emptyset$, on choisit une feuille $u \in F(T)$ quelconque, on met son unique voisin v dans la solution puis on supprime u et v de T (les arêtes incidentes à u et à v sont retirées de T et les degrés des sommets restants sont mis à jour).

Théorème 5. *Soit $T = (V, E)$ un arbre quelconque. L'algorithme* OT *retourne toujours une couverture optimale pour l'arbre T.*

Démonstration. Soit $T = (V, E)$ un arbre. Considérons une exécution quelconque de l'algorithme OT sur T. A chaque étape, il choisit une feuille u, il met son unique voisin v dans la solution, puis

il retire ces deux sommets de l'arbre. Il existe une exécution de l'algorithme ED (que nous avons présenté et décrit dans le chapitre 1 : voir l'algorithme 2 page 29) qui retourne exactement deux fois plus de sommets que OT. En effet, si l'on se base sur le même ordre d'apparition des sommets, ED effectue le même traitement que OT, à la seule différence qu'il met non seulement v dans la solution mais aussi u, ce qui génère au final une couverture qui contient deux fois plus de sommets. Or, comme ED est 2-approché, si OT retourne des solutions dont la taille est égale à la moitié de celles créées par ED, cela signifie qu'il est optimal. □

La figure 2.8 montre un exemple d'exécution de l'algorithme OT sur un arbre.

Propriétés générales

Dans ce qui va suivre, nous allons montrer, à l'aide de plusieurs étapes intermédiaires, qu'il n'existe pas de couverture optimale d'un arbre qui contienne toutes ses feuilles. Nous allons aussi montrer qu'il existe toujours une couverture optimale qui ne contient pas de feuille, pour un arbre constitué d'au moins trois sommets.

Un premier résultat intermédiaire énoncé dans le théorème 6 et que l'on va réutiliser par la suite est que l'algorithme OT est capable de retourner toutes les couvertures optimales possibles d'un arbre.

Théorème 6. *Soit $T = (V, E)$ un arbre quelconque. Pour n'importe quelle couverture optimale C^* de T, il existe une exécution de l'algorithme* OT *sur T qui retourne C^*.*

Démonstration. On procède par récurrence sur n, le nombre de sommets de T.

- **Cas de base.**

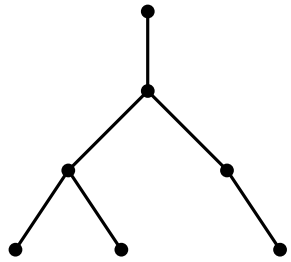
Pour $n = 1$. L'arbre T réduit à un sommet isolé n'a pas de feuille. Donc, OT retourne l'ensemble vide. Il s'agit là de l'unique solution possible.
Pour $n = 2$. L'arbre T réduit à deux sommets a exactement deux feuilles u et v, connectées par l'arête uv. Il existe deux solutions optimales : $C_a^* = \{u\}$ et $C_b^* = \{v\}$. Elles peuvent être retournées toutes les deux par OT. Pour C_a^*, il choisit v lors de la première étape ; pour C_b^*, il choisit d'abord u.
Pour $n = 3$. L'arbre T réduit à trois sommets a exactement deux feuilles, u et v, connectées à un sommet interne i (T est un chemin P_3). Il existe une solution optimale qui contient uniquement le nœud interne i, et cette solution est bien retournée par OT.

- **Cas général.**

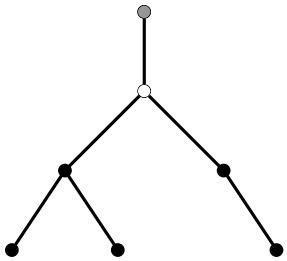
Pour $n > 3$.

Hypothèse de récurrence. Soit T un arbre avec n sommets et C^* une couverture optimale quelconque de T. C^* peut être construite par OT.

Récurrence. Soit T un arbre avec $n + 1$ sommets et C^* une couverture optimale quelconque de T. Nous allons montrer qu'il existe une exécution de l'algorithme OT sur T qui retourne C^*.



(a) Arbre sur lequel on exécute OT

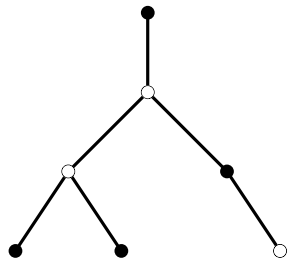


(b) Première étape : on choisit une feuille au hasard (sommet en gris), puis on place son voisin dans la solution

(c) Deuxième étape : on procède de la même façon que précédemment sur l'arbre restant, après avoir retiré le sommet gris et le sommet blanc de l'étape précédente

(d) Troisième étape : on procède de la même façon que précédemment

(e) Quatrième étape : il n'y a plus rien à faire (l'arbre restant ne contient plus d'arête)



(f) Résultat final obtenu sur l'arbre de la figure (a)

FIG. 2.8 – Exécution de l'algorithme OT sur un arbre à 7 sommets

Soit u une feuille de T et v son unique voisin. Il y a deux possibilités.

Première possibilité : $v \in C^*$. Dans ce cas, $u \notin C^*$ (sinon, C^* ne serait pas optimale). On note $T_1, T_2, \ldots, T_k$ les k sous-arbres de v (à l'exception de u).

Pour tout i compris entre 1 et k : $C_i^* = C^* \cap V(T_i)$ est une couverture optimale de T_i. Autrement

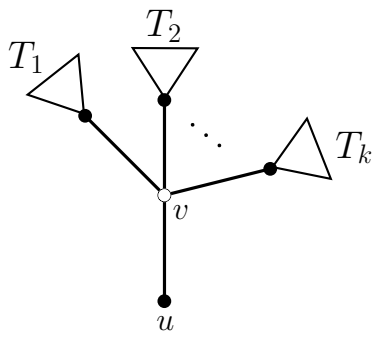


FIG. 2.9 – Vue d'ensemble de T, du point de vue de l'arête uv, lorsque $v \in C^*$

(supposons le contraire), il existe une couverture optimale $C_i^{*'}$ de T_i telle que $|C_i^{*'}| < |C_i^*|$. Dans ce cas, on aurait $C^{*'} = (C^* \setminus C_i^*) \cup C_i^{*'}$ une couverture de T et $|C^{*'}| < |C^*|$, ce qui contredit le fait que C^* est optimale pour T.

On donne une exécution de OT qui construit C^* :

***Etape* 1.** Choisir u dans un premier temps, ce qui fait mettre v dans C^* et retirer u et v de T.

***Etape* 2.** Maintenant, les T_i sont des arbres isolés (on a « coupé » leur racine v), les C_i^* sont optimales et $|T_i| < n$. On peut donc appliquer l'hypothèse de récurrence sur chacun des sous-arbres T_i.

Cette exécution globale de l'algorithme OT retourne bien au final C^*.

Seconde possibilité : $v \notin C^*$. Dans ce cas, tous les voisins de v sont dans C^*.

(a) Aucun T_i n'est réduit à un seul sommet v_i, autrement, $C^{*'} = (C^* \setminus \{v_i, u\}) \cup \{v\}$ serait une couverture de T et $|C^{*'}| < |C^*|$, ce qui contredit le fait que C^* est optimale.

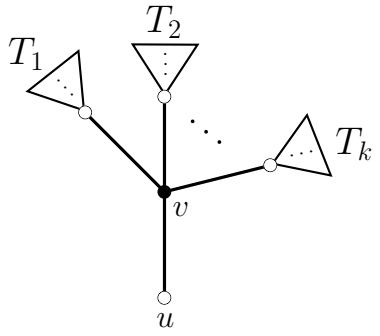


FIG. 2.10 – Vue d'ensemble de T, du point de vue de l'arête uv, lorsque $u \in C^*$

Pour tout i compris entre 1 et k : $C_i^* = C^* \cap V(T_i)$ et $|C_i^*| \geq 1$, (par la propriété *(a)*). C_i^* est une couverture optimale de T_i. Autrement (supposons le contraire), il existe une couverture optimale $C_i^{*'}$ de T_i telle que $|C_i^{*'}| < |C_i^*|$. Construisons une nouvelle couverture pour T. On remplace les sommets de C_i^* par les sommets (structurellement moins nombreux) de $C_i^{*'}$. On retire u de $C^{*'}$ et

on met v dans la nouvelle couverture. On a alors

$$C^{*'} \;=\; (C^* \setminus C_i^* \setminus \{u\}) \cup C_i^{*'} \cup \{v\} \;\; .$$

1. $C^{*'}$ est bien une couverture de T puisque toutes les arêtes de T_i sont couvertes, ainsi que toutes les arêtes incidentes à v (car $v \in C^{*'}$) et toutes les autres arêtes (leur couverture n'a pas changé).
2. $|C^{*'}| < |C^*|$ car $|C_i^{*'}| < |C_i^*|$ et le changement « de u vers v » ne modifie pas le nombre de sommets de C^*.

Ces propriétés contredisent le fait que C^* est optimale pour T.

On donne une exécution de OT qui construit C^*.

***Etape** i.* On applique OT sur T_i : il retourne C_i^*. On peut appliquer l'hypothèse de récurrence, puisque C_i^* est une couverture optimale de T_i et $|T_i| < n$.

Etape** $k+1$ **(à la fin). Il reste une arête uv sur laquelle on applique OT : il retourne u (le voisin de la feuille choisie v sur cette dernière arête).

Cette exécution globale de l'algorithme OT retourne bien au final C^*. □

Maintenant, nous allons utiliser ce résultat pour montrer qu'il n'existe pas de couverture optimale qui contienne toutes les feuilles d'un arbre.

Proposition 1. *Pour tout arbre $T = (V, E)$ quelconque, l'algorithme* OT *ne retourne jamais une couverture contenant toutes les feuilles de T.*

Démonstration. Soit u la première feuille considérée par l'algorithme OT lors de la première étape. Son unique voisin v est mis dans la solution, puis u et v sont retirés de l'arbre. Ces deux sommets ne sont plus considérés par la suite. Par conséquent, il existe au moins une feuille (en l'occurrence u) qui n'est pas retournée par OT. □

Corollaire 1. *Pour tout arbre $T = (V, E)$, il n'existe pas de couverture optimale contenant toutes les feuilles de T.*

Démonstration. Le résultat est directement déduit du théorème 6 et de la proposition 1 : OT, qui est capable de retourner toutes les couvertures optimales d'un arbre, ne retourne jamais toutes ses feuilles. □

Proposition 2. *Pour n'importe quel arbre $T = (V, E)$ avec $n > 2$ sommets, il existe une couverture optimale qui ne contient pas de feuille.*

Démonstration. Soit $T = (V, E)$ un arbre à $n > 2$ sommets. Soit C^* une couverture optimale de T. Supposons que C^* contienne des feuilles de T (d'après le corollaire 1, elle ne peut pas contenir toutes ses feuilles). Soit F_C l'ensemble des feuilles de T contenues dans C^*. Considérons un sommet $u \in F_C$. Soit v son voisin. On sait que C^* est minimale pour l'inclusion (puisqu'elle est optimale).

(a) Exemple d'arbre avec une couverture optimale

(b) Modification de la couverture optimale : on remplace les feuilles (le sommet gris) par leurs voisins

FIG. 2.11 – Construction d'une couverture optimale sans feuille pour un arbre

Par conséquent, $v \notin C^*$ (u couvre uniquement l'arête uv). On peut alors substituer chaque feuille $u \in F_C$ par son voisin v dans la solution. Cette opération ne change pas la taille de C^*. Un exemple est donné par la figure 2.11. □

Propriétés de ASLL

En nous basant sur les résultats précédents, nous allons montrer que l'algorithme ASLL n'est pas capable de retourner une couverture optimale sur des arbres constitués d'au moins trois sommets.

Lemme 2. *Pour tout arbre étiqueté $T = (V, L, E)$ à $n > 2$ sommets, l'algorithme* ASLL *retourne toujours une couverture qui contient l'ensemble des feuilles de T.*

Démonstration. Soit $T = (V, L, E)$ un arbre étiqueté à $n > 2$ sommets. Toutes les feuilles de T possèdent chacune exactement un voisin de plus fort degré. En effet, il ne peut y avoir d'arête uv dans T telle que u et v soient des feuilles puisque T est, par définition, connexe. Par conséquent, ASLL retourne toutes les feuilles de T. □

Théorème 7. *Pour tout arbre étiqueté $T = (V, L, E)$ à $n > 2$ sommets, l'algorithme* ASLL *ne retourne jamais une couverture optimale pour T.*

Démonstration. D'après le lemme 2, l'algorithme ASLL retourne toujours toutes les feuilles d'un arbre à $n > 2$ sommets. Mais, selon le corollaire 1, il n'existe pas de couverture optimale qui contienne toutes les feuilles d'un arbre. Par conséquent, ASLL ne retourne jamais une solution optimale pour des arbres de taille $n > 2$. □

Propriétés de SLL

Proposition 3. *Pour tout arbre étiqueté $T = (V, L, E)$ à $n > 2$ sommets, l'algorithme* SLL *retourne toujours une couverture qui ne contient aucune feuille de T.*

Démonstration. Dans un arbre à au moins trois sommets, les feuilles (de degré 1) ne peuvent avoir de voisin de degré strictement inférieur (dans un graphe, aucun sommet ne peut avoir de voisin de degré 0), ni de même degré. Par conséquent, les couvertures retournées par l'algorithme SLL pour des arbres de taille $n > 2$ ne contiennent jamais de feuilles. □

Bien que l'algorithme SLL ne retourne jamais les feuilles d'un arbre dans les solutions qu'il construit, cela ne signifie pas pour autant qu'il est optimal. En effet, il existe des arbres pour lesquels, quel que soit l'étiquetage des sommets, il ne pourra jamais retourner de solution optimale. La figure 2.12 montre un exemple d'un tel arbre. Sur cet arbre, SLL retourne les sommets blancs (la couverture optimale), ainsi que le sommet gris (qui ne fait pas partie de la solution optimale), puisque ce dernier possède des voisins de plus petit degré.

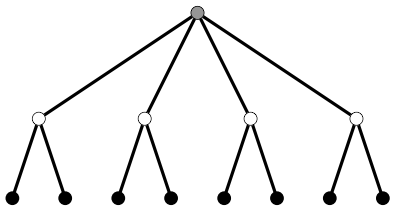


FIG. 2.12 – Arbre pour lequel l'algorithme SLL ne peut pas retourner de solution optimale

Un premier résultat pour SLL et ASLL sur les arbres

Par rapport à leurs comportements vis-à-vis des feuilles (ASLL les retourne toutes, tandis que SLL n'en retourne aucune), nous pouvons énoncer un premier résultat comparatif à propos de ces deux algorithmes sur les arbres.

Corollaire 2. *Soit $T = (V, L, E)$ un arbre étiqueté quelconque et contenant au moins $\frac{n}{2} \geq 2$ feuilles. Soit $\mathsf{SLL}(T)$ (resp. $\mathsf{ASLL}(T)$) la taille de la couverture construite par l'algorithme* SLL *(resp.* ASLL*) exécuté sur T. On a $\mathsf{SLL}(T) \leq \mathsf{ASLL}(T)$.*

Démonstration. D'après le lemme 2 et la proposition 3, quel que soit l'étiquetage des sommets de T, l'algorithme SLL ne retourne jamais de feuille, tandis qu'à l'inverse, l'algorithme ASLL retourne toutes les feuilles de T. On a donc $\mathsf{SLL}(T) \leq n - \frac{n}{2}$ et $\mathsf{ASLL}(T) \geq \frac{n}{2}$, ce qui donne bien $\mathsf{SLL}(T) \leq \mathsf{ASLL}(T)$. □

Les algorithmes SLL et ASLL sur les chemins

Nous avons aussi montré que ASLL ne pouvait pas retourner une solution meilleure que SLL sur des chemins, c'est-à-dire sur des arbres particuliers contenant uniquement deux feuilles.

Théorème 8. *Soit $P_n = (V, L, E)$ un chemin étiqueté quelconque à $n > 2$ sommets. Soit $\mathsf{SLL}(P_n)$ (resp. $\mathsf{ASLL}(P_n)$) la taille de la couverture construite par l'algorithme* SLL *(resp.* ASLL*) exécuté sur*

P_n. On a

$$\mathsf{ASLL}(P_n) \quad = \quad \mathsf{SLL}(P_n) + 1 \ . \tag{2.4}$$

En d'autres termes, quel que soit le chemin étiqueté de taille $n > 2$ sur lequel ils sont exécutés, SLL *retourne toujours une solution meilleure que* ASLL.

Démonstration. Soit $P_n = (V, L, E)$ un chemin étiqueté de taille $n > 2$. On peut utiliser des expressions régulières pour démontrer (2.4).
Soit $\Sigma = \{L, M, R\}$ un alphabet (c'est-à-dire un ensemble de symboles), dans lequel :

- L représente un sommet quelconque de P_n qui est retourné par ASLL mais pas par SLL. Ce sommet n'a pas de voisin de degré inférieur, ni de voisin droit de même degré.
- M représente un sommet quelconque de P_n qui est retourné par ASLL et par SLL. Il possède au moins un voisin de degré inférieur ou un voisin droit de même degré et il possède au moins un voisin de degré supérieur ou un voisin gauche de même degré.
- R représente un sommet quelconque de P_n qui est retourné par SLL mais pas par ASLL. Ce sommet n'a pas de voisin de degré supérieur, ni de voisin gauche de même degré.

A l'exception des sommets isolés (que l'on ne considère pas ici), il n'y a pas d'autres choix possibles pour les sommets : ils sont soit de type L, soit de type R, ou bien de type M.

Soit $\mathcal{E}_P$ l'expression régulière définie sur Σ pour le chemin étiqueté P_n. Cette expression représente le parcours du chemin, en allant d'une extremité à une autre (« de gauche à droite » ou bien « de droite à gauche »). Elle modélise, dans l'ordre, les types de sommets rencontrés. On obtient

$$\mathcal{E}_P \quad = \quad L(M^*RM^*L)^+ \tag{2.5}$$

Dans la parenthèse de (2.5), on a le même nombre de L que de R. Par conséquent, si l'on considère la totalité de $\mathcal{E}_P$, il y a exactement un L de plus par rapport au nombre de R.

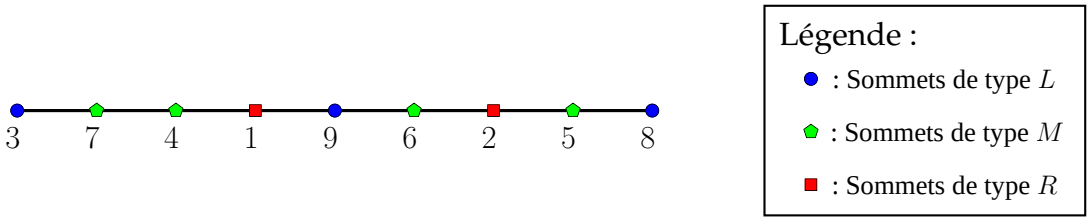


FIG. 2.13 – Exemple de chemin étiqueté pour lequel on exhibe les trois classes de sommets

Montrons que l'expression régulière $\mathcal{E}_P$ est juste, c'est-à-dire qu'elle simule bien le parcours d'un chemin P_n d'une extrémité à l'autre. Elle vérifie les propriétés suivantes.

1. On a bien $\mathcal{E}_P$ qui commence par L et qui se termine par L. En effet, les deux extrémités d'un chemin sont des feuilles, qui sont retournées par ASLL mais pas par SLL.

2. On ne peut pas avoir de séquences RR ou bien LL dans $\mathcal{E}_P$. En effet, soit $uv \in E$ une arête quelconque de P_n. Dans cette arête, sans perte de généralité[2], si u est voisin gauche de v, alors v est voisin droit de u (voir la définition 4 page 36).
3. L'existence de séquences RM^+R ou bien LM^+L dans $\mathcal{E}_P$ est impossible, sinon, cela signifierait qu'il existe un sommet u de type M tel que $d(u) > 2$, ce qui n'est pas possible dans un chemin.

On a donc bien exactement un L supplémentaire par rapport au nombre de R dans $\mathcal{E}_P$. Donc, $\mathsf{ASLL}(P_n) = \mathsf{SLL}(P_n) + 1$. □

Nous avons donc prouvé que l'algorithme SLL retournait toujours une solution meilleure que l'algorithme ASLL sur les chemins.

Et sur les arbres en général ?

Nous sommes parvenus à montrer que ASLL ne retourne jamais une solution meilleure que SLL sur des chemins étiquetés de taille $n > 2$ et sur des arbres étiquetés constitués d'au moins $\frac{n}{2} \geq 2$ feuilles. Nous sommes convaincus que ce résultat est valable sur l'ensemble des arbres étiquetés (c'est l'objet de la conjecture 1).

Conjecture 1. *Soit $T = (V, L, E)$ un arbre étiqueté quelconque. Soit $\mathsf{SLL}(T)$ (resp. $\mathsf{ASLL}(T)$) la taille de la couverture retournée par l'algorithme SLL (resp. ASLL) pour T. On a*

$$\mathsf{ASLL}(T) \geq \mathsf{SLL}(T) .$$

A ce jour, nous n'avons pas trouvé d'arbre qui contredirait cette conjecture.

2.3 Etude de la complexité en nombre de requêtes

Dans cette section, nous nous focaliserons sur la complexité des trois algorithmes. Une telle étude est pertinente dans le sens où les temps d'exécution des algorithmes, exécutés sur de grandes instances de données, peuvent varier de manière significative.

2.3.1 Description de la notion de requête

Lorsque nous avons décrit les trois algorithmes, nous avons vu que, durant leurs exécutions, l'unité de traitement récupérait les sommets un par un, dans n'importe quel ordre des labels (pas forcément de 1 jusqu'à n). Nous avons aussi vu que les voisins d'un sommet u étaient eux aussi récupérés un par un, dans un ordre quelconque. Cela implique deux situations.

1. Si u n'est pas envoyé dans la couverture lors de l'examen de son voisin courant, l'unité de calcul récupère un autre voisin de u, qui n'a pas encore été scanné. S'il n'y a plus de voisin à

[2]Ce principe de parité peut s'énoncer aussi avec les voisins de degrés inférieurs ou les voisins droits de même degré et les voisins de degrés supérieurs ou les voisins gauches de même degré.

scanner (lorsque u a été comparé avec tous ses voisins), elle décide alors de manière définitive que u ne fait pas partie de la solution.

2. Si u est envoyé dans la couverture à la suite de sa comparaison avec son voisin courant, alors le système n'a pas besoin de récupérér ses voisins restants (s'il en reste). Par conséquent, l'unité de traitement n'est pas obligée de récupérer tous les voisins d'un sommet. Elle peut « sauter » les voisins restants, à partir du moment où elle a pris une décision définitive pour le sommet u (en l'occurrence, l'ajouter à la solution).

A partir de ces précisions, nous pouvons maintenant donner la définition de la notion de requête, telle qu'on l'entend.

Définition 7 (Requête)**.** *On appelle* requête *le fait de récupérer, pour un sommet, un voisin (son label et son degré, si besoin) qui n'a pas encore été scanné.*

Dans cette section, nous allons évaluer le *nombre de requêtes* effectuées par les trois algorithmes pour construire leurs solutions. Pour un graphe étiqueté, ce nombre dépend uniquement de l'ordre dans lequel les voisins des sommets sont récupérés.

Dans notre modèle, nous supposons que l'unité de calcul met plus de temps à récupérer un voisin (à partir d'un graphe stocké à distance) qu'à comparer deux sommets (leurs labels et leurs degrés, si besoin) stockés sur sa mémoire. De ce fait, le nombre de requêtes est un facteur déterminant pour évaluer les temps d'exécution des algorithmes.

Notons que ce type d'étude est proche des études réalisées dans l'approche *query complexity* [102]. De plus, elle a une granularité plus fine que l'analyse de complexité effectuée pour les algorithmes I/O-efficient [128]. (Pour plus de précisions, se reporter à la section 1.3.)

Si les graphes étaient récupérés arête par arête, il n'y aurait pas grand intérêt à étudier le nombre de requêtes car, de façon générale, il serait toujours égal à m. A l'inverse, tel que nous les parcourons, il peut y avoir des différences notables d'une exécution à l'autre. De plus, le fait de scanner les graphes sommet par sommet permet, bien souvent, d'améliorer cette complexité en nombre de requêtes. Nous allons voir par la suite que les performances moyennes en terme de complexité en nombre de requêtes des trois algorithmes étudiés sont souvent inférieures à m.

Notations supplémentaires

Avant d'étudier de manière fine le nombre de requêtes effectuées par les algorithmes LL, SLL et ASLL, nous avons besoin de formaliser quelques notions déjà évoquées auparavant.

Définition 8. *Soit $G = (V, L, E)$ un graphe étiqueté quelconque. Soit $u \in V$ un sommet de G.*

- *Soit $d^+(u)$ (resp. $d^-(u)$) le nombre de* voisins droits *(resp.* gauches*) de u.*
- *Soit $d_{\text{inf}}(u)$ (resp. $d_{\text{sup}}(u)$) le nombre de* voisins de degré inférieur *(resp.* supérieur*) de u.*
- *Soit $\sigma^+(u)$ (resp. $\sigma^-(u)$) le nombre de* voisins droits *(resp.* gauches*) de u ayant* le même degré.

2.3.2 Bornes inférieures et supérieures sur le nombre de requêtes

Dans cette sous-section, nous allons donner, pour les trois algorithmes, les bornes minimales et maximales sur le nombre de requêtes qu'ils effectuent.

Pour chaque critère, nous donnerons tout d'abord dans un lemme intermédiaire les formules exactes qui s'appliquent à un graphe étiqueté, puis nous donnerons les formules exactes en moyenne sur l'ensemble des étiquetages d'un graphe, en supposant qu'ils sont équiprobables.

Nombre minimum de requêtes

Lemme 3. *Soit $G = (V, L, E)$ un graphe étiqueté quelconque. Soit C_{A} la couverture construite par l'algorithme A exécuté sur G. On désigne par $\mathcal{B}\{\mathtt{q}_{\mathsf{A}}(G, L)\}$ le nombre minimum de requêtes effectuées par l'algorithme A durant son exécution sur G. De façon générale, pour LL, SLL et ASLL, on a*

$$\mathcal{B}\{\mathtt{q}_{\mathsf{A}}(G, L)\} \quad = \quad \sum_{u \notin C_{\mathsf{A}}} d(u) + |C_{\mathsf{A}}| \ . \tag{2.6}$$

Démonstration. Soit $G = (V, L, E)$ un graphe étiqueté. Soit C_{A} une couverture construite par l'algorithme A pour G. Considérons un sommet u de G. Dans le meilleur des cas, quand $u \in C_{\mathsf{A}}$, on le met dans la solution au bout d'une requête (pour LL, c'est le cas si l'on récupère en premier un voisin droit). Lorsque $u \notin C_{\mathsf{A}}$, on doit récupérer tous ses voisins pour décider de manière définitive qu'il ne sera pas dans la solution (pour LL, c'est le cas s'il ne possède aucun voisin droit). □

Théorème 9. *Soit $G = (V, E)$ un graphe quelconque. Soit*

$$\mathcal{B}\{\mathtt{q}_{\mathsf{A}}(G)\} \quad = \quad \min_{L \in \mathbb{L}(G)} \mathcal{B}\{\mathtt{q}_{\mathsf{A}}(G, L)\}$$

le nombre minimum de requêtes effectuées par l'algorithme A durant son exécution sur G. Pour l'algorithme LL, on a

$$\mathcal{B}\{\mathtt{q}_{\mathsf{LL}}(G)\} \quad = \quad n - c + \sum_{G_i \in \mathcal{CC}(G)} \delta_i \ , \tag{2.7}$$

où $\mathcal{CC}(G)$ désigne l'ensemble des composantes connexes de G (de cardinalité c) et δ_i le degré minimal de la composante connexe $G_i \subseteq G$.

Démonstration. Considérons tout d'abord un graphe connexe (avec $c = 1$). Nous utiliserons le raisonnement qui va suivre pour des graphes à plusieurs composantes connexes.

Soit $G = (V, E)$ un graphe connexe quelconque. Soit $P \subset V$ un sous-ensemble quelconque des sommets de G (P n'est pas forcément une couverture de G), de taille au plus $n - 1$. On peut alors appliquer (2.6) en remplaçant C_{A} par P. Notons $B(P)$ la valeur obtenue. Il s'agit de trouver la plus petite valeur de $B(P)$ parmi tous les sous-ensembles P d'au plus $n - 1$ sommets. Précisons les choses, en prenant un sous-ensemble P qui a k sommets (avec $k \leq n - 1$).

$$B(P) \quad = \quad \sum_{u \notin P} d(u) + k \ .$$

Si $k < n-1$, on peut diminuer (pas forcément strictement) cette valeur en transférant dans P un sommet u qui n'en faisait pas partie. En effet, comme G n'a aucun sommet isolé (il est connexe), $d(u) > 0$. Ainsi, $|P|$ gagne une unité, tandis que la somme des degrés perd au moins une unité. Donc, au total, on diminue potentiellement la valeur de $B(P)$ (en tout cas, on ne l'augmente pas).

Supposons maintenant que seuls des sous-ensembles P de taille $k < n-1$ minimisent la somme $B(P)$. Avec ce qui précède, chaque sommet ajouté permet de diminuer potentiellement la valeur de $B(P)$. Ainsi, en ajoutant des sommets à P jusqu'à en avoir $n-1$ (notons P' l'ensemble obtenu), on obtient $B(P') \leq B(P)$ et $|P'| = n-1$, ce qui contredit l'hypothèse de départ. On peut donc rechercher l'ensemble P qui minimise $B(P)$ seulement parmi les ensembles à $n-1$ sommets. Or, il est facile de voir que parmi tous ces ensembles à $n-1$ sommets, celui qui exclut un sommet de degré minimum est associé à la plus petite valeur de $B(P)$, car ils ont tous un $B(P) = n-1+d(u)$ où u est le seul sommet n'appartenant pas à P.

Nous avons raisonné sur des sous-ensembles de sommets plus grands que des couvertures. Or, d'après le théorème 3, il existe bien une couverture C des sommets de G constructible par LL et contenant tous ses sommets sauf un, de degré δ (il suffit que ce sommet soit étiqueté par la plus grande valeur, à savoir n). Ainsi, il existe une solution C ayant une valeur $B(C)$ qui est égale au minimum des valeurs $B(P)$ pour tous les sous-ensembles de sommets P de taille quelconque, qu'ils soient ou non des couvertures.

On ne peut donc pas trouver de couverture C ayant une valeur $B(C)$ plus petite que $n-1+\delta$. Cette valeur est atteinte pour n'importe quel graphe connexe G (il suffit de prendre un étiquetage L tel que LL retourne tous les sommets sauf un de degré minimum).

On peut au final appliquer ce raisonnement sur chacune des c composantes connexes $G_i \in \mathcal{CC}(G)$ de G. Soit n_i le nombre de sommets de la composante G_i. On a donc

$$\begin{aligned} \mathcal{B}\{\mathfrak{q}_{\mathsf{LL}}(G)\} &= \sum_{G_i \in \mathcal{CC}(G)} (n_i - 1 + \delta_i) \\ &= n - c + \sum_{G_i \in \mathcal{CC}(G)} \delta_i \ . \end{aligned}$$

□

Corollaire 3. *Le nombre minimum de requêtes effectuées par l'algorithme* LL *exécuté sur un graphe* G *connexe (en considérant l'ensemble de ses étiquetages* $\mathbb{L}(G)$*) est*

$$\mathcal{B}\{\mathfrak{q}_{\mathsf{LL}}(G)\} = n - 1 + \delta \ . \tag{2.8}$$

De plus, pour tout graphe G *quelconque, on a* $\mathcal{B}\{\mathfrak{q}_{\mathsf{A}}(G)\} \geq n-1+\delta$*, avec* $\mathsf{A} = \mathsf{LL}$*,* SLL *ou* ASLL*.*

Démonstration. Le résultat est déduit directement du théorème 9 et du théorème 4. □

Sur des graphes de faible densité (où $m \in \Theta(n)$), le nombre minimum de requêtes effectuées par les algorithmes peut être supérieur à m (c'est le cas sur les arbres). En revanche, sur des graphes de forte densité (où $m \in \mathcal{O}(n^2)$), ce nombre minimum peut être bien plus petit que m.

Nombre maximum de requêtes

Lemme 4. *Soit $G = (V, L, E)$ un graphe étiqueté quelconque. Soit C_{A} la couverture construite par l'algorithme* A *pour G. On désigne par $\mathcal{W}\{\mathtt{q}_{\mathsf{A}}(G, L)\}$ le nombre maximum de requêtes effectuées par l'algorithme* A *durant son exécution sur G. Pour* LL*, on a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{LL}}(G, L)\} = \sum_{u \notin C_{\mathsf{LL}}} d(u) + \sum_{u \in C_{\mathsf{LL}}} (d^-(u) + 1) \ . \tag{2.9}$$

Pour l'algorithme SLL*, on a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{SLL}}(G, L)\} = \sum_{u \notin C_{\mathsf{SLL}}} d(u) + \sum_{u \in C_{\mathsf{SLL}}} (d_{\sup}(u) + \sigma^-(u) + 1) \ . \tag{2.10}$$

Et pour ASLL*, on a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{ASLL}}(G, L)\} = \sum_{u \notin C_{\mathsf{ASLL}}} d(u) + \sum_{u \in C_{\mathsf{ASLL}}} (d_{\inf}(u) + \sigma^+(u) + 1) \ . \tag{2.11}$$

Démonstration. Nous donnons la preuve pour LL. Celles pour SLL et ASLL sont similaires.

Soit $G = (V, L, E)$ un graphe étiqueté. Soit C_{LL} une couverture construite par LL pour G. Considérons un sommet u de G. On sait que $u \in C_{\mathsf{LL}}$ si et seulement si il possède au moins un voisin droit. Dans le pire des cas, l'unité de traitement récupère tous les voisins gauches de u avant d'obtenir un premier voisin droit. De ce fait, elle effectue exactement $d^-(u) + 1$ requêtes avant de mettre u dans la solution. Autrement, si $u \notin C_{\mathsf{LL}}$, alors on doit récupérer tous ses voisins gauches avant de décider finalement qu'il n'est pas dans la solution (puisque l'on ne sait pas *a priori* si un sommet a des voisins droits ou pas), ce qui génère exactement $d(u)$ requêtes. □

Théorème 10. *Soit $G = (V, E)$ un graphe quelconque. Soit*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{A}}(G)\} = \max_{L \in \mathbb{L}(G)} \mathcal{W}\{\mathtt{q}_{\mathsf{A}}(G, L)\}$$

le nombre maximum de requêtes effectuées par l'algorithme A *durant son exécution sur G. On a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{A}}(G)\} = m + |C_{\mathsf{A}}^{\max}| \ , \tag{2.12}$$

où $|C_{\mathsf{A}}^{\max}|$ est la taille maximale d'une couverture retournée par l'algorithme A *pour le graphe G (lorsque l'on considère l'ensemble de ses étiquetages possibles $\mathbb{L}(G)$).*

Démonstration. Nous donnons les preuves pour LL et pour SLL. Celle pour ASLL est similaire à celle pour SLL.

Preuve pour LL. Soit $G = (V, L, E)$ un graphe étiqueté et C_{LL} la couverture construite par LL pour G. On peut simplifier (2.9) de la manière suivante.

$$\sum_{u \notin C_{\mathsf{LL}}} d(u) + \sum_{u \in C_{\mathsf{LL}}} (d^-(u) + 1) = m + |C_{\mathsf{LL}}| \tag{2.13}$$

puisque, $\forall u \notin C_{\mathsf{LL}}$, $d(u) = d^-(u)$ et $\sum_{u \in V} d^-(u) = m$. Maintenant, si l'on maximise (2.13) en considérant tous les $n!$ étiquetages possibles de $\mathbb{L}(G)$, cela revient à maximiser la taille de C_{LL}. De ce fait, on obtient bien $\mathcal{W}\{\mathtt{q}_{\mathsf{LL}}(G)\} = m + |C_{\mathsf{LL}}^{\max}|$.

Preuve pour SLL. Soit $G = (V, L, E)$ un graphe étiqueté et C_{SLL} la couverture construite par SLL exécuté sur G. On peut simplifier (2.10) de la manière suivante.

$$\sum_{u \notin C_{\mathsf{SLL}}} d(u) + \sum_{u \in C_{\mathsf{SLL}}} (d_{\sup}(u) + \sigma^-(u) + 1) \quad = \quad m + |C_{\mathsf{SLL}}| \tag{2.14}$$

puisque, $\forall u \notin C_{\mathsf{SLL}}$, $d(u) = d_{\sup}(u) + \sigma^-(u)$ et $\sum_{u \in V}(d_{\sup}(u) + \sigma^-(u)) = m$. Maintenant, si l'on maximise (2.14) en considérant tous les $n!$ étiquetages possibles de $\mathbb{L}(G)$, cela revient à maximiser la taille de C_{SLL}. Par conséquent, on obtient $\mathcal{W}\{\mathtt{q}_{\mathsf{SLL}}(G)\} = m + |C_{\mathsf{SLL}}^{\max}|$. □

Corollaire 4. *Le nombre maximum de requêtes effectuées par l'algorithme* LL *sur un graphe G quelconque (en considérant l'ensemble de ses étiquetages $\mathbb{L}(G)$) ayant c composantes connexes est*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{LL}}(G)\} \quad = \quad m + n - c \ . \tag{2.15}$$

De plus, $\mathcal{W}\{\mathtt{q}_{\mathsf{A}}(G)\} \ \leq \ m + n - 1$.

Démonstration. Les résultats pour LL sont déduits du théorème 10 et du lemme 4. □

2.3.3 Analyse de la complexité moyenne

Dans cette sous-section, nous donnerons les formules exactes exprimant le nombre moyen de requêtes effectuées par les trois algorithmes pour construire une couverture. Pour cela, nous supposerons que pour chaque sommet $u \in V$ d'un graphe $G = (V, E)$, ses $d(u)$ voisins peuvent être récupérés dans l'un des $d(u)!$ ordres possibles, en considérant que ces $d(u)!$ ordres apparaissent chacun avec une probabilité uniforme égale à $\frac{1}{d(u)!}$.

Nous donnerons dans un premier temps les formules exactes exprimant le nombre moyen de requêtes effectuées sur un graphe étiqueté, puis nous généraliserons ce résultat en considérant tous les $n!$ étiquetages d'un graphe de manière équiprobable.

Lemme 5. *Soit $G = (V, L, E)$ un graphe étiqueté quelconque. Soit C_{A} la couverture construite par l'algorithme* A *pour G. On désigne par $\mathbb{E}\big[\mathtt{q}_{\mathsf{A}}(G, L)\big]$ le nombre moyen de requêtes effectuées par l'algorithme* A *durant son exécution sur G. Pour* LL*, on a*

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(G, L)\big] \quad = \quad \sum_{u \notin C_{\mathsf{LL}}} d(u) + \sum_{u \in C_{\mathsf{LL}}} \frac{d(u) + 1}{d^+(u) + 1} \ . \tag{2.16}$$

Pour l'algorithme SLL*, on a*

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(G, L)\big] \quad = \quad \sum_{u \notin C_{\mathsf{SLL}}} d(u) + \sum_{u \in C_{\mathsf{SLL}}} \frac{d(u) + 1}{d_{\inf}(u) + \sigma^+(u) + 1} \ . \tag{2.17}$$

Et pour ASLL*, on a*

$$\mathbb{E}\big[\mathfrak{q}_{\mathsf{ASLL}}(G,L)\big] \;=\; \sum_{u\notin C_{\mathsf{ASLL}}} d(u) + \sum_{u\in C_{\mathsf{ASLL}}} \frac{d(u)+1}{d_{\sup}(u)+\sigma^-(u)+1}\ . \tag{2.18}$$

Démonstration. Nous donnons la preuve pour LL. Celles pour SLL et ASLL sont similaires.

Soit $G=(V,L,E)$ un graphe étiqueté quelconque. Soit C_{LL} une couverture construite par LL pour G. Considérons un sommet u de G. Si $u\notin C_{\mathsf{LL}}$, alors l'unité de traitement doit récupérer tous les $d(u)$ voisins de u. Autrement, elle effectue

$$\frac{d(u)+1}{d^+(u)+1}$$

requêtes en moyenne avant de récupérer le premier des $d^+(u)$ voisins droits de u.

L'obtention de cette valeur s'explique de la manière suivante. Imaginons un sac contenant a boules blanches et b boules noires. A chaque étape, on pioche une boule, sans la remettre dans le sac, et on s'intéresse au nombre d'étapes avant de piocher la première boule noire. Il est possible de montrer que le nombre moyen de tirages effectués avant de tomber sur la première boule noire est égal à

$$\frac{a}{b+1}\ .$$

Une preuve simple, didactique et originale de ce résultat est donnée par *M. Glaymann* dans [66].

Maintenant, si l'on suppose que les $d(u)$ voisins de u sont des boules dans un sac, avec $d^-(u)$ boules blanches et $d^+(u)$ boules noires, alors, avec $a=d^-(u)$ et $b=d^+(u)$, on obtient

$$\frac{a}{b+1}+1 \;=\; \frac{d^-(u)}{d^+(u)+1}+1 \;=\; \frac{d(u)+1}{d^+(u)+1}$$

requêtes en moyenne (incluant celle qui donne la première « boule noire »). □

Théorème 11. *Soit $G=(V,E)$ un graphe quelconque. Soit*

$$\mathbb{E}\big[\mathfrak{q}_{\mathsf{A}}(G)\big] \;=\; \frac{1}{n!}\sum_{L\in\mathbb{L}(G)} \mathbb{E}\big[\mathfrak{q}_{\mathsf{A}}(G,L)\big]$$

le nombre moyen de requêtes effectuées par l'algorithme A *durant son exécution sur G, en considérant tous les étiquetages de $\mathbb{L}(G)$ de manière équiprobable. Pour* LL*, on a*

$$\mathbb{E}\big[\mathfrak{q}_{\mathsf{LL}}(G)\big] \;=\; \sum_{u\in V} H(d(u))\ , \tag{2.19}$$

où $H(n)=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ désigne le nombre harmonique *d'ordre n, avec $H(0)=0$.*

Pour l'algorithme SLL*, on a*

$$\begin{aligned}\mathbb{E}\big[\mathfrak{q}_{\mathsf{SLL}}(G)\big] \;=\;& \sum_{u\in V}\frac{d(u)+1}{\sigma(u)+1}\big(H(d_{\inf}(u)+\sigma(u)+1)-H(d_{\inf}(u))\big)\\ &-\sum_{u\mid d_{\inf}(u)=0}\frac{1}{\sigma(u)+1}\ .\end{aligned} \tag{2.20}$$

Et pour ASLL, *on a*

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(G)\big] &= \sum_{u\in V}\frac{d(u)+1}{\sigma(u)+1}\big(H(d_{\sup}(u)+\sigma(u)+1)-H(d_{\sup}(u))\big)\\ &- \sum_{u|d_{\sup}(u)=0}\frac{1}{\sigma(u)+1}\ .\end{aligned} \tag{2.21}$$

Démonstration. Nous donnons les preuves pour LL et pour SLL. La preuve pour ASLL est similaire à celle pour SLL.

Preuve pour LL. Soit $G=(V,E)$ un graphe quelconque. On détermine la *contribution* de chaque sommet $u\in V$ dans $\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(G)\big]$. Soit $L\in\mathbb{L}(G)$ un étiquetage de G et C_{LL} la couverture construite par l'algorithme LL exécuté sur le graphe G étiqueté par L. Notons que pour chaque sommet $u\in V$, $u\notin C_{\mathsf{LL}}$ si et seulement si $d^+(u)=0$.

Soit $\beta_k(u)$ la proportion d'étiquetages $L\in\mathbb{L}(G)$ pour lesquels $d^+(u)=k$. En appliquant (2.16), la *contribution* du sommet u dans $\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(G)\big]$ est de

$$\beta_0(u)\cdot d(u)+\sum_{k=1}^{d(u)}\beta_k(u)\cdot\frac{d(u)+1}{k+1}\ . \tag{2.22}$$

Calculons la valeur de $\beta_k(u)$. La valeur de $d^+(u)$ dépend uniquement de l'étiquetage du sommet u par rapport à l'étiquetage de ses voisins. Il y a exactement

$$\binom{n}{d(u)+1}\cdot d(u)!\times(n-(d(u)+1))! \tag{2.23}$$

étiquetages pour lesquels $d^+(u)=k$. En effet, cela revient à étiqueter les sommets de G de la manière suivante. Premièrement, on choisit $d(u)+1$ labels parmi n et on attribue à u le $(k+1)^{\text{ème}}$ plus grand label, dans le but d'avoir $d^+(u)=k$. Il reste alors $d(u)!$ possibilités pour étiqueter les voisins de u et $(n-(d(u)+1))!$ possibilités pour étiqueter les autres sommets de G. On obtient donc $\beta_k(u)$ en divisant (2.23) par $n!$. Donc, $\forall k\in\{0,\dots,d(u)\}$, $\beta_k(u)=\frac{1}{d(u)+1}$.

Maintenant, on simplifie (2.22), ce qui donne

$$\begin{aligned}\beta_0(u)\cdot d(u)+\sum_{k=1}^{d(u)}\beta_k(u)\cdot\frac{d(u)+1}{k+1} &= \frac{d(u)}{d(u)+1}+\sum_{k=1}^{d(u)}\frac{1}{k+1}\\ &= \frac{d(u)}{d(u)+1}+\sum_{k=2}^{d(u)+1}\frac{1}{k}\ =\ 1+\sum_{k=2}^{d(u)}\frac{1}{k}\ =\ H(d(u))\ .\end{aligned}$$

Preuve pour SLL. Soit $G=(V,E)$ un graphe quelconque. On détermine la *contribution* de chaque sommet $u\in V$ dans $\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(G)\big]$. Soit $L\in\mathbb{L}(G)$ un étiquetage de G et C_{SLL} la couverture construite par l'algorithme SLL exécuté sur le graphe G étiqueté par L. Notons que pour chaque sommet $u\in V$, $u\notin C_{\mathsf{SLL}}$ si et seulement si $d_{\inf}(u)=0$ et $\sigma^+(u)=0$. Aussi, notons que si $d_{\inf}(u)>0$, quel que soit l'étiquetage des sommets de G, u est toujours sélectionné par SLL.

Soit $\beta'_k(u)$ la proportion des étiquetages $L\in\mathbb{L}(G)$ pour lesquels $\sigma^+(u)=k$.

1. Si $d_{\inf}(u) > 0$, quelle que soit la valeur de $\sigma^+(u)$ (comprise entre 0 et $\sigma(u)$ et désignée par k par la suite), la *contribution* du sommet u dans $\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(G)\big]$ est de

$$\beta'_k(u) \cdot \frac{d(u)+1}{d_{\inf}(u)+k+1} \ . \tag{2.24}$$

2. Si $d_{\inf}(u) = 0$, alors le fait que u est ou n'est pas dans la solution retournée par SLL dépend uniquement de son étiquetage par rapport à l'étiquetage de ses voisins de même degré. Dans ce cas, sa *contribution* dans $\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(G)\big]$ est de

$$\begin{cases} \beta'_k(u) \cdot \frac{d(u)+1}{k+1} & \text{avec } k = 1, \dots, \sigma(u) \quad \text{si } u \in C_{\mathsf{SLL}}, \\ \beta'_0(u) \cdot d(u) & \text{sinon.} \end{cases} \tag{2.25}$$

On obtient la valeur de $\beta'_k(u)$ en appliquant le même raisonnement que pour LL. On remplace $d(u)$ par $\sigma(u)$ dans $\beta_k(u)$ et donc, on a, pour chaque sommet u, $\beta'_k(u) = \frac{1}{\sigma(u)+1}$, $\forall k \in \{0, \dots, \sigma(u)\}$.

Dans le but de simplifier (2.24), on applique ce résultat pour chaque sommet u tel que $d_{\inf}(u) > 0$:

$$\begin{aligned} \sum_{k=0}^{\sigma(u)} \frac{1}{\sigma(u)+1} \cdot \frac{d(u)+1}{d_{\inf}(u)+k+1} &= \frac{d(u)+1}{\sigma(u)+1} \sum_{k=1}^{\sigma(u)+1} \frac{1}{d_{\inf}(u)+k} \\ &= \frac{d(u)+1}{\sigma(u)+1} \big(H(d_{\inf}(u)+\sigma(u)+1) - H(d_{\inf}(u)) \big) \ , \end{aligned} \tag{2.26}$$

et on applique ce résultat sur (2.25), pour chaque sommet u tel que $d_{\inf}(u) = 0$:

$$\begin{aligned} \frac{d(u)}{\sigma(u)+1} + \sum_{k=1}^{\sigma(u)} \frac{1}{\sigma(u)+1} \cdot \frac{d(u)+1}{k+1} &= \frac{d(u)}{\sigma(u)+1} + \frac{d(u)+1}{\sigma(u)+1} \left(\sum_{k=1}^{\sigma(u)+1} \frac{1}{k} - 1 \right) \\ &= \frac{-1}{\sigma(u)+1} + \frac{d(u)+1}{\sigma(u)+1} \sum_{k=1}^{\sigma(u)+1} \frac{1}{k} \\ &= \frac{d(u)+1}{\sigma(u)+1} \cdot H(\sigma(u)+1) - \frac{1}{\sigma(u)+1} \ . \end{aligned} \tag{2.27}$$

Le résultat final est obtenu en effectuant la somme de (2.26) et de (2.27) pour tous les sommets du graphe G. □

Corollaire 5. *Le nombre moyen de requêtes effectuées par les algorithmes* LL, SLL *et* ASLL *sur un graphe Δ-régulier quelconque est $n \cdot H(\Delta)$, ce qui tend vers $n \cdot \log \Delta$ lorsque Δ tend vers $+\infty$.*

Démonstration. Comme dans G on a $d(u) = \Delta$ pour tout $u \in V$, le résultat pour LL découle immédiatement. On a aussi $d_{\inf}(u) = d_{\sup}(u) = 0$ et $\sigma(u) = d(u)$. En appliquant ces différentes valeurs, on peut simplifier (2.20) et obtenir les résultats pour SLL et ASLL. □

Exemples d'application sur des familles de graphes

Comme dans la section précédente, les formules présentées et décrites ici dans le théorème 11 permettent de calculer en temps polynomial $\mathbb{E}\big[\mathtt{q}_{\mathsf{A}}(G)\big]$ pour un graphe G lorsque $\mathsf{A} = \mathsf{LL}$, SLL ou ASLL.

Afin de tenter d'établir des relations entre ces trois algorithmes, nous avons étudié le nombre de requêtes effectuées en moyenne sur quelques familles de graphes connues.

Sur les étoiles. Soit $S_n = (V, E)$ une étoile à n sommets. Appliquons respectivement (2.19), (2.20) et (2.21) sur S_n. Pour tout $n > 2$, sachant que dans une étoile S_n telle que $n > 2$, aucun sommet ne possède un voisin de même degré, on obtient

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(S_n)\big] \;=\; H(n-1) + n - 1 \;.$$

Pour SLL, les $n-1$ feuilles de S_n ne possèdent pas de voisin de plus petit degré. A l'inverse, le centre de S_n a $n-1$ voisins (les feuilles) qui ont un plus petit degré. Donc, on a

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(S_n)\big] \;=\; (n-1)\cdot 2 + n\big(H(n) - H(n-1)\big) - (n-1) \;=\; n \;.$$

Pour ASLL, le centre de S_n n'a pas de voisin de plus fort degré. Par contre, chaque feuille possède un voisin (le centre) qui a un degré plus grand. Donc, on a

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(S_n)\big] \;=\; n + (n-1)\cdot 2\big(H(2) - H(1)\big) - 1 \;=\; 2n - 2 \;.$$

Donc, on peut facilement voir que

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(S_n)\big] \;<\; \mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(S_n)\big] \;<\; \mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(S_n))\big] \;.$$

Sur les graphes bipartis complets. Soit $K_{a,b} = (X \cup Y, E)$ un graphe biparti complet avec $n = a + b$ sommets (où $a = |X|$ et $b = |Y|$). Admettons que $a > b > 4$. Dans un graphe biparti complet $K_{a,b}$ tel que $a \neq b$, aucun sommet ne possède un voisin de même degré. On a donc

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(K_{a,b})\big] \;=\; a\cdot H(b) + b\cdot H(a) \;.$$

Pour SLL, les a sommets de X n'ont pas de voisin de plus petit degré. En revanche, chaque sommet de Y possède a voisins (ceux de X) ayant un degré plus grand. Donc, on obtient

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(K_{a,b})\big] \;=\; a(b+1) + b(a+1)\big(H(a+1) - H(a)\big) - a \;=\; ab + b \;.$$

Pour ASLL, les b sommets de Y ne possèdent pas de voisin ayant un degré plus grand qu'eux. Par contre, chaque sommet de X a b voisins (ceux de Y) de plus fort degré. Par conséquent, on a

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(K_{a,b})\big] \;=\; b(a+1) + a(b+1)\big(H(b+1) - H(b)\big) - b \;=\; ab + a \;.$$

On peut facilement voir que SLL est meilleur que ASLL (parce que $a > b$). Comparons LL et SLL.

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(K_{a,b})\big] - \mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(K_{a,b})\big] &= ab + b - a \cdot H(b) - b \cdot H(a) \\ &= \frac{ab}{2} - a \cdot H(b) + \frac{ab}{2} - b \cdot H(a) + b \ > \ 0\end{aligned}$$

car $\frac{b}{2} > H(b)$ lorsque $b > 4$ et $\frac{a}{2} > H(a)$ quand $a > 4$.

Par conséquent, lorsque $a > b > 4$, LL produit en moyenne moins de requêtes que SLL et ASLL sur les graphes bipartis complets.

Définition 9 (Colliers)**.** *Soit $CK_{l,w} = (V, E)$ un* collier *avec $n = l \times w$ sommets. Un collier $CK_{l,w}$ est un cycle constitué de l graphes complets, appelés* perles*, où chaque graphe complet K_i ($i \in \{1, \ldots, l\}$) a w sommets : $w - 2$ sommets de degré $w - 1$ et 2 sommets distincts a_i et b_i, de degré w, appelés* connecteurs*, qui relient K_i à son prédécesseur ($b_i a_{i-1} \in E$) et à son successeur ($a_i b_{i+1} \in E$) dans le cycle.*

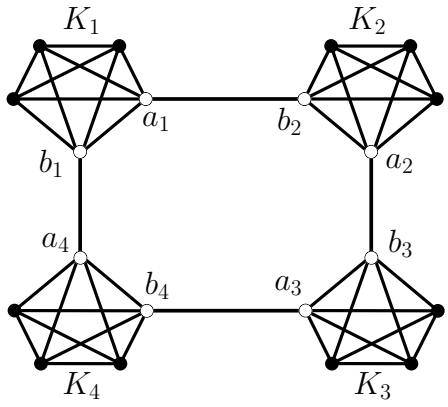


FIG. 2.14 – Collier $CK_{4,5}$ à 4 perles, constituées chacune de 5 sommets

Sur les colliers. Soit $CK_{l,w} = (V, E)$ un collier avec $n = l \times w$ sommets. Dans un collier $CK_{l,w}$, chaque sommet appartenant aux l graphes complets (à l'exception des $2l$ connecteurs a_i et b_i) possèdent $w - 3$ voisins de même degré. Quant aux connecteurs, ils possèdent chacun 2 voisins de même degré. Admettons que $l > 1$ et que $w > 4$. On a

$$\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(CK_{l,w})\big] \ = \ l(w-2) \cdot H(w-1) + 2l \cdot H(w) \ = \ n \cdot H(w-1) + \frac{2l}{w} \ .$$

Pour SLL, les $l(w-2)$ sommets autres que les connecteurs n'ont pas de voisin de plus faible degré. quant aux connecteurs, ils possèdent $w - 2$ voisins (les sommets du graphe complet dans lequel ils

sont) ayant un degré plus petit qu'eux. On a donc

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(CK_{l,w})\big] &= l(w-2)\cdot\frac{w}{w-2}\cdot H(w-2) + 2l\cdot\frac{w+1}{3}\big(H(w+1)-H(w-2)\big) - \frac{l(w-2)}{w-2}\\ &= n\cdot H(w-2) + \frac{2l}{3}\cdot\frac{3w^2-1}{w(w-1)} - l\ .\end{aligned}$$

Pour ASLL, les $2l$ connecteurs n'ont pas de voisin de plus fort degré. Cependant, les autres sommets possèdent chacun 2 voisins (les connecteurs issus du même graphe complet) ayant un degré plus grand qu'eux. Par conséquent, on a

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(CK_{l,w})\big] &= 2l\cdot\frac{w+1}{3}\cdot H(3) + l(w-2)\cdot\frac{w}{w-2}\big(H(w)-H(2)\big) - \frac{2l}{3}\\ &= n\cdot H(w) - \frac{5n}{18} + \frac{5l}{9}\ .\end{aligned}$$

Comparons ces résultats.

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{LL}}(CK_{l,w})\big] - \mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(CK_{l,w})\big] &= \frac{5n}{18} + \frac{2l}{w} - \frac{14l}{9}\\ &= \frac{l(5w^2-28w+36)}{18w} \ > \ 0\end{aligned}$$

lorsque $w > 3$, car le polynôme $5w^2 - 28w + 36$ a deux racines : 2 et 3,6 ; et il est positif $\forall w \leq 2$ et $\forall w \geq 3{,}6$.

$$\begin{aligned}\mathbb{E}\big[\mathtt{q}_{\mathsf{SLL}}(CK_{l,w})\big] - \mathbb{E}\big[\mathtt{q}_{\mathsf{ASLL}}(CK_{l,w})\big] &= \frac{5n}{18} + \frac{n}{w-1} - \frac{2l}{3w(w-1)} - \frac{23l}{9}\\ &= \frac{l(5w^3-33w^2+46w-12)}{18w(w-1)} \ > \ 0\end{aligned}$$

lorsque $w > 4$, car le polynôme $5w^3 - 33w^2 + 46w - 12$ a trois racines : 0,34 ; 1,48 et 4,78 ; et il est positif $\forall w \in [0{,}34; 1{,}48]$ et $\forall w \geq 4{,}78$.

Par conséquent, lorsque $l > 1$ et lorsque $w > 4$, ASLL produit en moyenne moins de requêtes que LL et SLL sur les colliers.

Synthèse des résultats obtenus sur d'autres familles de graphes. En plus des trois exemples d'application détaillés ci-dessus, de la même manière, nous avons déterminé de façon analytique le nombre de requêtes effectuées en moyenne par les trois algorithmes sur d'autres familles de graphes. Le tableau 2.2 fournit une synthèse de ces résultats.

Nous constatons que pour chaque algorithme, il existe au moins une famille de graphes pour laquelle il effectue, en moyenne, moins de requêtes que les deux autres.

Mis à part sur les chemins et les étoiles, pour les autres familles de graphes exhibées dans le tableau 2.2, les ordres obtenus entre les trois algorithmes ne sont pas les mêmes que pour les qualités moyennes des solutions retournées, ce qui rend leur analyse plus difficile.

Algorithmes / Familles de graphes	LL	SLL	ASLL
Graphes Δ-réguliers	1^{er} *ex aequo*	1^{er} *ex aequo*	1^{er} *ex aequo*
Chemins	$2^{\text{ème}}$	1^{er}	$3^{\text{ème}}$
Etoiles	$2^{\text{ème}}$	1^{er}	$3^{\text{ème}}$
Graphes bipartis complets	1^{er}	$2^{\text{ème}}$	$3^{\text{ème}}$
Graphes d'*Avis-Imamura*	1^{er}	$2^{\text{ème}}$	$3^{\text{ème}}$
Graphes d'*Avis-Imamura* modifiés	1^{er}	$2^{\text{ème}}$	$3^{\text{ème}}$
Graphes d'*Avis-Imamura* étendus	1^{er}	$2^{\text{ème}}$	$3^{\text{ème}}$
Grilles	$3^{\text{ème}}$	1^{er}	$2^{\text{ème}}$
Spider graphs	$3^{\text{ème}}$	1^{er}	$2^{\text{ème}}$
Colliers	$3^{\text{ème}}$	$2^{\text{ème}}$	1^{er}

TAB. 2.2 – Synthèse des résultats obtenus sur les comparaisons du nombre de requêtes effectuées en moyenne par les trois algorithmes sur certaines familles de graphes

2.4 Bilan

Nous avons étudié dans ce chapitre trois algorithmes, LL, SLL et ASLL pour le problème du VERTEX COVER, qui satisfont les contraintes de notre modèle : ils ne modifient pas le graphe en entrée, ils n'utilisent quasiment pas de mémoire sur l'unité de traitement (ils sont *memory-efficient*) et ils n'ont pas besoin d'accéder à la solution (pour la lire ou pour la modifier) durant leurs exécutions. Ils sont donc adaptés pour construire une couverture sur un graphe de grande taille avec une machine standard.

Nous avons tout d'abord comparé la qualité des solutions retournées par ces trois méthodes. Nous avons dans un premier temps donné les formules exactes exprimant les tailles moyennes des solutions qu'elles construisent, ce qui nous a permis de montrer qu'aucune des trois ne pouvait être élue comme la meilleure sur l'ensemble des graphes.

Après avoir montré que pour tout graphe, il existait toujours un étiquetage des sommets tel que LL retournait une couverture optimale, nous avons dans un second temps analysé les comportements des heuristiques SLL et ASLL sur les arbres. Nous avons mis l'accent sur le fait que SLL y était globalement plus performant que ASLL, en prouvant notamment que ASLL ne retournait jamais de couverture optimale sur les arbres.

Par la suite, nous nous sommes intéressés à la complexité en nombre de requêtes de ces trois algorithmes. Après en avoir fourni les bornes minimales et maximales, nous avons donné les formules exactes exprimant le nombre moyen de requêtes qu'ils effectuent. Là encore, nous avons montré qu'aucun des trois ne pouvait être élu comme le meilleur.

Les formules analytiques fournies dans ce chapitre peuvent aider un utilisateur à choisir un algorithme adapté, en fonction des informations dont il dispose sur les graphes à traiter (certains domaines d'application sont liés à certains types de graphes spécifiques) et suivant le critère qu'il

cherche à optimiser (s'il doit calculer une solution rapidement, il choisira l'algorithme qui génère le moins de requête).

S'il l'on ne dispose d'aucune information sur les graphes à traiter et si l'on cherche à optimiser à la fois la qualité des solutions construites et la complexité en nombre de requêtes, il est préférable d'utiliser l'algorithme SLL. En effet, même s'il est très mauvais sur quelques classes de graphes particulières, il offre de bonnes performances en moyenne sur la plupart des graphes.

Cependant, l'algorithme LL est aussi intéressant car il dispose d'une « marge de manœuvre » maximale pour tout graphe et ce sur les deux critères. De plus, il est plus simple à mettre en œuvre puisqu'il n'a pas besoin de récupérer les degrés des voisins scannés. En effet, dans certaines implémentations, cela peut augmenter les temps d'exécutions.

L'algorithme ASLL doit, quant à lui, se cantonner à des utilisations bien spécifiques, lorsque l'on dispose de suffisamment d'informations sur les graphes à traiter et lorsque l'on cherche à optimiser un seul critère. En effet, sur les graphes d'*Avis-Imamura* étendus par exemple, il est le meilleur en terme de qualité de solution mais il est le plus mauvais en terme de complexité en nombre de requêtes.

Une première perspective de ce chapitre est de prouver la conjecture 1, qui dit que SLL est meilleur que ASLL sur les arbres. En effet, nous avons réussi à montrer que SLL était meilleur que ASLL sur les chemins et sur les arbres dont au moins la moitié des sommets sont des feuilles. Nous sommes convaincus que ce résultat est valable sur l'ensemble des arbres.

Une autre perspective serait de compléter les résultats que l'on a obtenu en moyenne sur les deux critères étudiés, en déterminant les formules (exactes, si possible) des variances, afin d'évaluer de manière encore plus fine les performances des trois algorithmes, et en déterminant leurs rapports d'approximation en moyenne pour l'ensemble des graphes.

Pour aller plus loin dans l'étude menée dans ce chapitre, on pourrait s'intéresser à d'autres problèmes proches du VERTEX COVER (tels que la version pondérée par exemple) et tenter d'y adapter les trois heuristiques.

Chapitre 3

Etude d'algorithmes à n bits mémoire pour le problème du Vertex Cover sur de grandes instances

Dans ce chapitre, nous allons assouplir la contrainte C_2 de notre modèle décrit dans le chapitre 1 page 14, ce qui va nous permettre d'implémenter et d'étudier d'autres algorithmes.

Après avoir rappelé notre modèle de traitement et fourni quelques éléments utiles à la description des algorithmes que nous allons étudier, nous adapterons l'algorithme OT, que nous avons décrit dans le chapitre 2 et qui est optimal sur les arbres. Nous étudierons sa complexité en nombre de passes (il peut en effectuer plusieurs) et en nombre de requêtes. Nous adapterons ensuite les algorithmes LR et ED (que nous avons décrits dans la section 1.4) et nous étudierons leurs complexités en nombre de requêtes. Enfin, nous présenterons, adapterons et étudierons un autre algorithme : Pitt. Nous étudierons sa complexité, puis nous nous intéresserons à la quantité de mémoire qu'il utilise.

3.1 Extension de notre modèle

Par rapport au modèle que l'on a considéré jusqu'à présent, nous supposons désormais que l'unité de traitement dispose d'un espace mémoire de n bits. Redonnons les contraintes liées à notre modèle.

C_1. Les données fournies en entrées ne doivent pas être modifiées : l'*intégrité* de l'instance doit être préservée.

C_2'. L'unité de traitement dispose d'un espace mémoire de n bits.

C_3. Les éléments de la solution doivent être envoyés « à la volée », c'est-à-dire à chaque fois qu'ils sont ajoutés à la solution.

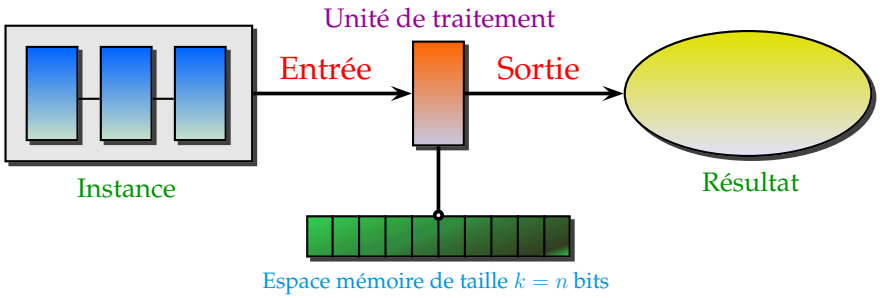


FIG. 3.1 – Vue d'ensemble de notre modèle avec n bits d'espace mémoire

La contrainte C_2 (devenue C_2') a subi une légère modification. En effet, jusqu'à présent, nous avons supposé qu'un nombre constant d'éléments étaient stockables sur l'unité de traitement. Désormais, nous supposerons qu'un nombre d'éléments linéaire en fonction du nombre de sommets sont stockables en mémoire.

Typiquement, chacun des n bits correspond à une valeur booléenne reliée à un sommet du graphe.

Le graphe est toujours stocké sous la forme d'une liste d'adjacence et la notion de graphes étiquetés est toujours présente. Cependant, dans les résultats que nous présenterons, nous parlerons simplement de graphes puisque, contrairement à LL, SLL et ASLL, les labels vont uniquement être utilisés dans les implémentations détaillées des algorithmes : ils ne seront pas employés par les algorithmes dans leurs prises de décisions (ils ne serviront pas par exemple à comparer les sommets entre eux).

Dans ce qui va suivre, nous donnerons les descriptions détaillées des algorithmes que nous avons adaptés dans ce modèle. Pour cela, nous devons au préalable introduire quelques notions.

Tout d'abord, nous supposons que le nombre de sommets n est connu à l'avance (il peut par exemple être récupéré au début de l'exécution des algorithmes). Dans les opérations que nous définissons et qui vont être utilisées par les différents algorithmes implémentés, il y a des opérations de communication et des opérations internes. Les opérations de communication servent à récupérer le graphe morceau par morceau et à envoyer les éléments de la solution « à la volée », là aussi morceau par morceau (nous rappelons que l'instance traitée en entrée et que la solution produite en sortie sont stockées sur des éléments distants : voir la figure 3.1). Les opérations internes servent à manipuler l'espace mémoire de n bits disponible sur l'unité de traitement.
L'unité de traitement dispose donc des opérations suivantes.

- **Les opérations de communication**

Soit $G = (V, E)$ le graphe stocké à distance. On a :

- $u \leftarrow getVertex(i)$ permet de récupérer le sommet u (son label $L(u)$ et son degré $d(u)$) situé à la $i^{\text{ème}}$ position dans la liste d'adjacence ;
- $v \leftarrow getNeighbor(u, j)$ permet de récupérer le $j^{\text{ème}}$ voisin v (son label $L(v)$ et son degré $d(v)$) du sommet u. Cette opération correspond à une *requête* (voir la définition 7 page 56).

Soit C la couverture construite durant l'exécution d'un algorithme. C peut être implémentée sous différentes formes (ensemble de sommets, flux de données, *etc.*). L'unité de traitement ne connaît pas forcément la manière dont est stockée la solution à construire. Nous mettons donc à sa disposition une opération générique d'ajout de sommet. Nous nous assurerons par la suite que les algorithmes que nous implémenterons ne produiront pas de doublons. On a donc :

– $u \rightarrow C$ permet d'ajouter le sommet u à la couverture C.

Les algorithmes que nous étudions génèrent les solutions « à la volée » : une fois qu'un sommet est placé dans la couverture, il ne peut plus en être retiré. Il n'y a donc pas besoin de définir une opération de retrait.

• **Les opérations internes**

Soit Mem l'espace mémoire de n bits disponible sur l'unité de traitement. Il s'agit d'un tableau de n bits consécutifs. On a :

– $Mem.init(val)$ permet d'initialiser tous les bits de Mem à la valeur val (bien souvent, nous les initialiserons à 0) ;
– $Mem[i]$ permet de lire le $i^{\text{ème}}$ bit du tableau Mem.
– $Mem[i] \leftarrow 1$ permet de mettre à 1 le $i^{\text{ème}}$ bit de Mem.

3.2 Adaptation de l'algorithme OT

Revenons sur cet algorithme, que nous avons déjà présenté dans le chapitre 2 (l'algorithme 11 page 47 en fournit une version générique).

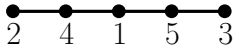
Contrairement à ce que nous allons faire par la suite pour les autres algorithmes que nous détaillerons, ici, nous utilisons les n bits disponibles sur l'unité de traitement pour stocker/marquer les sommets que l'on « retire » de l'arbre. En effet, le fait de garder en mémoire la trace des sommets supprimés nous permet de calculer les degrés des sommets dans l'arbre induit par la suppression des sommets déjà marqués. Cela nous permet donc de toujours trouver des feuilles et ainsi d'amener l'exécution de OT à son terme.
L'implémentation 1 donne une version détaillée de l'adaptation de OT dans notre modèle. La figure 3.2 en donne un exemple d'exécution détaillé.

Jusqu'à présent, nous avons étudié des algorithmes qui parcouraient l'ensemble des sommets exactement une fois. Nous allons voir que l'algorithme OT, tel que nous l'avons implémenté, a parfois besoin de parcourir plusieurs fois la liste des sommets du graphe pour construire une solution. En effet, il se peut que toutes les arêtes ne soient pas couvertes après une première lecture de la liste d'adjacence. Il faut donc s'assurer que l'algorithme retourne bien une couverture dans tous les cas. Par conséquent, à la fin de chaque passe, lorsqu'on a traité le $n^{\text{ème}}$ sommet, on regarde s'il reste des sommets non marqués dans l'arbre. Si c'est le cas, cela signifie qu'il reste potentiellement des arêtes à couvrir. Il peut aussi cependant n'y avoir que des sommets isolés non encore marqués. Ce cas est possible puisque lorsque l'on marque un sommet u, les degrés de ses voisins dans l'arbre induit diminuent. Certains deviennent parfois isolés, et s'ils ont été traités avant u dans la liste

Implémentation 1 : OT

Données : un arbre étiqueté $T = (V, L, E)$ à n sommets
Résultat : une couverture $C \subseteq V$ de T

```
début
    Mem.init(0)
    i ← 1
    tant que i ≤ n faire
        u ← getVertex(i)
        si Mem[L(u)] = 0 alors
            /* On calcule le degré du sommet après suppression d'arêtes.      */
            j ← 1
            deg ← 0
            tant que j ≤ d(u) ∧ deg ≤ 1 faire
                /* On scanne l'arête uv.                                      */
                v ← getNeighbor(u, j)
                si Mem[L(v)] = 0 alors
                    deg ← deg + 1
                    w ← v
                j ← j + 1
            si deg = 0 alors Mem[L(u)] = 1
            si deg = 1 alors
                w → C
                /* Suppression des arêtes couvertes.                          */
                Mem[L(u)] ← 1
                Mem[L(w)] ← 1
        si i = n alors
            /* On teste s'il reste des arêtes à couvrir.                      */
            j ← 1
            trouve ← faux
            tant que j ≤ n ∧ ¬trouve faire
                si Mem[j] = 0 alors trouve ← vrai
                sinon j ← j + 1
            /* Si c'est le cas, alors on refait une passe.                    */
            si trouve alors i ← 1 sinon i ← i + 1
        sinon i ← i + 1
fin
```



(a) Arbre sur lequel on exécute l'algorithme OT

$1 \to 4, 5$
$2 \to 4$
$3 \to 5$
$4 \to 2, 1$
$5 \to 1, 3$

(b) Liste d'adjacence de l'arbre, parcourue de haut en bas et de gauche à droite par OT

$Mem[1]$	$Mem[2]$	$Mem[3]$	$Mem[4]$	$Mem[5]$
0	0	0	0	0

i	j	deg
2	3	2
$C = \emptyset$		

(c) Première étape : état de la mémoire après le traitement du sommet 1

$Mem[1]$	$Mem[2]$	$Mem[3]$	$Mem[4]$	$Mem[5]$
0	1	0	1	0

i	j	deg
3	2	1
$C = \{4\}$		

(d) Deuxième étape : état de la mémoire après le traitement du sommet 2

$Mem[1]$	$Mem[2]$	$Mem[3]$	$Mem[4]$	$Mem[5]$
0	1	1	1	1

i	j	deg
4	2	1
$C = \{4, 5\}$		

(e) Troisième étape : état de la mémoire après le traitement du sommet 3

$Mem[1]$	$Mem[2]$	$Mem[3]$	$Mem[4]$	$Mem[5]$
0	1	1	1	1

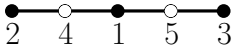
i	j	deg	$trouve$
1	1	1	vrai
$C = \{4, 5\}$			

(f) Quatrième étape : état de la mémoire après le traitement des sommets 4 et 5

$Mem[1]$	$Mem[2]$	$Mem[3]$	$Mem[4]$	$Mem[5]$
1	1	1	1	1

i	j	deg	$trouve$
6	6	0	faux
$C = \{4, 5\}$			

(g) Cinquième étape : état de la mémoire à la fin de la seconde passe



(h) Couverture obtenue à la fin de l'exécution

FIG. 3.2 – Exécution de l'algorithme OT (détaillé par l'implémentation 1) sur un chemin à 5 sommets (seules les étapes où $Mem[L(u)] = 0$ sont détaillées)

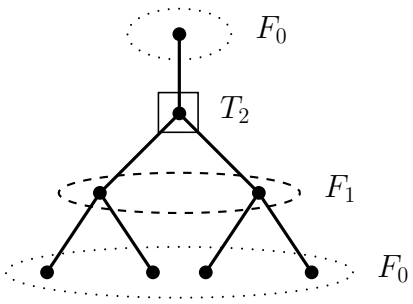


FIG. 3.3 – Découpage des sommets d'un arbre en couches

d'adjacence, ils ne pourront être marqués que lors de la prochaine passe.

Quelle que soit la passe et le sommet, dans tous les cas, on calcule le degré courant du sommet traité, en tenant compte de ses voisins déjà marqués. En effet, s'il s'agit d'une « véritable » feuille (pour laquelle un simple test sur le degré nous renseigne), nous n'effectuons de toute façon qu'une seule requête qui nous sert à récupérer son voisin et à le placer dans la solution.

Dans notre modèle, nous rappelons que les sommets sont stockés les uns à la suite des autres dans un ordre quelconque, l'unité de traitement les récupérant dans l'ordre dans lequel ils sont stockés sur la machine externe. Ainsi, si l'algorithme effectue plusieurs passes, l'ordre dans lequel les sommets apparaissent est le même pour chaque passe. Nous allons construire, à l'aide de la définition 10, des ordres de stockage pour montrer des bornes sur le nombre de passes et le nombre de requêtes effectuées par l'algorithme OT.

Définition 10 (Couches de sommets d'un arbre)**.** *Soit $T = (V, E)$ un arbre quelconque. Soit $F(T)$ l'ensemble des feuilles de T. De base, on a $T_0 = T$ et $F_0 = F(T)$. Par construction, on a $T_{i+1} = T_i - F_i$ et $F_{i+1} = F(T_{i+1})$, pour i allant de 0 à γ (on s'arrête à l'étape j, lorsque $T_{j+1} = \emptyset$: on obtient ainsi $\gamma = j$). On découpe l'arbre T en $\gamma + 1$ couches de sommets Ψ_i de la manière suivante : $\forall i \in \{0, \ldots, \gamma - 1\}$, $\Psi_i = F_i$ et $\Psi_\gamma = T_\gamma$. On note ω_i le nombre de sommets contenus dans la $i^{ème}$ couche de sommets ($\omega_i = |\Psi_i|$).*

La figure 3.3 montre un exemple de découpage des sommets d'un arbre en couches.

On remarque que la dernière couche de sommets, appelée de façon générale Ψ_γ, est soit constituée d'un sommet isolé (c'est le cas dans l'arbre de la figure 3.3), soit de deux sommets reliés par une arête (c'est le cas par exemple dans un chemin constitué de quatre sommets).

Dans ce qui va suivre, nous allons étudier la complexité en nombre de passes et la complexité en nombre de requêtes de l'algorithme OT.

3.2.1 Etude du nombre de passes

Nous allons montrer que l'algorithme OT est capable, sous certaines conditions, de n'effectuer qu'une seule passe pour construire une solution pour tout arbre. Nous allons aussi donner une borne supérieure sur le nombre maximum de passes qu'il peut réaliser.

Théorème 12. *Soit $T = (V, E)$ un arbre quelconque. Il existe un ordre de traitement des sommets de T pour lequel l'algorithme* OT *n'effectue qu'une seule passe.*

Démonstration. Soit $T = (V, E)$ un arbre à n sommets. Nous allons construire un ordre de traitement des sommets de T pour lequel l'algorithme OT n'effectue qu'une seule passe. On numérote les sommets de 1 à n, en procédant couche par couche (voir la définition 10). On attribue aux sommets de Ψ_0 les numéros allant de 1 à ω_0. Ensuite, on attribue aux sommets de Ψ_1 les numéros allant de $\omega_0 + 1$ à $\omega_0 + \omega_1$, *etc.* jusqu'à ce que tous les sommets soient numérotés (ceux de Ψ_γ sont numérotés de $\sum_{i=0}^{\gamma-1} \omega_i + 1$ à n). Ainsi, OT va regarder dans l'ordre les feuilles de T_0, puis celles de T_1, *etc.* pour finir par les sommets de T_γ. Par conséquent, à la fin de la première passe, il n'y aura plus de sommets à regarder dans l'arbre puisque chaque feuille scannée par OT est toujours marquée à la fin de son traitement. □

Théorème 13. *Soit $T = (V, E)$ un arbre quelconque, constitué de $\gamma + 1$ couches de sommets. Le nombre maximum de passes que peut effectuer l'algorithme* OT *sur T est de $\left\lceil \frac{\gamma+1}{2} \right\rceil$. Il existe des arbres pour lesquels cette borne n'est pas atteinte.*

Démonstration. Soit $T = (V, E)$ un arbre à n sommets et constitué de $\gamma + 1$ couches de sommets. De façon générale, quel que soit l'ordre dans lequel l'algorithme OT récupère les sommets de T, il marque au minimum deux couches de sommets lors d'une passe (sauf éventuellement à la dernière). En effet, lors de la $j^{\text{ème}}$ passe (avec $j \geq 1$), dans le pire des cas, OT traite les sommets de $\Psi_{2(j-1)}$ en dernier. Il les marque, ainsi que leurs voisins (qu'il met dans la solution). Or, d'après la définition 10, $\Psi_{2(j-1)+1} \subseteq \{u \mid \forall v \in \Psi_{2(j-1)}, uv \in E\}$. De ce fait, OT marque lors de la $j^{\text{ème}}$ passe au minimum les sommets de $\Psi_{2(j-1)}$ et de $\Psi_{2(j-1)+1}$ (il ne peut pas faire moins). Aussi, lors de la dernière passe, si T est constitué d'un nombre pair de couches de sommets, OT en marque au minimum deux, à savoir $\Psi_{\gamma-1}$ et Ψ_γ ; autrement, si $\gamma + 1$ est impair, il ne reste que les sommets de Ψ_γ à traiter lors de la dernière passe (il s'agit soit d'une arête, soit d'un sommet isolé). Par conséquent, l'algorithme OT ne peut pas effectuer plus de $\left\lceil \frac{\gamma+1}{2} \right\rceil$ passes, et ce quel que soit l'ordre dans lequel il récupère les sommets de l'arbre.

Montrons à présent que cette borne n'est pas toujours atteinte. Comme nous l'avons vu précédemment, les voisins des sommets d'une couche Ψ_i sont au minimum les sommets de Ψ_{i+1}. En effet, un sommet $u \in \Psi_i$ peut très bien avoir un voisin $v \in \Psi_{\tau>i+1}$. Par exemple, dans l'arbre de la figure 3.4, on peut voir que le sommet 12, qui appartient à Ψ_0, possède un voisin (le sommet 1) qui appartient à Ψ_5. Ce sommet est donc marqué dès la première passe, et ce quel que soit l'ordre dans lequel OT récupère les sommets (d'après le raisonnement établi précédemment, il devrait être marqué lors de la troisième passe). De ce fait, les degrés courants de ses autres voisins (ici les sommets 2 et 3) diminuent. Ainsi, lors de la seconde passe, OT marque les sommets de la couche Ψ_2 (les sommets 6 et 7) et de la couche Ψ_3 (les sommets 4 et 5), mais aussi ceux de la couche Ψ_4 (les sommets 2 et 3), qui sont devenus des feuilles après la suppression du sommet 1 lors de la précédente passe. A la fin de la seconde passe, tous les sommets sont déjà marqués. Par conséquent,

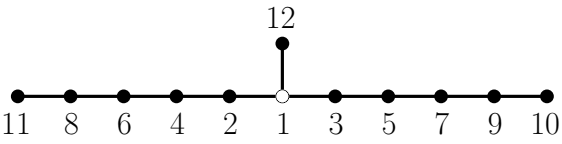


FIG. 3.4 – Arbre pour lequel l'algorithme OT ne peut effectuer un maximum de $\left\lceil \frac{\gamma+1}{2} \right\rceil$ passes

l'algorithme OT n'effectue pas de troisième passe sur cet arbre (qui compte pourtant cinq couches de sommets). □

Nous avons remarqué qu'il y avait une corrélation entre le diamètre d'un l'arbre et le nombre de passes effectuées par OT. En effet, si l'on considère les deux familles d'arbres qui présentent respectivement un diamètre minimum et maximum, à savoir les étoiles et les chemins, sur les étoiles, on ne peut pas effectuer plus d'une passe, tandis que sur les chemins, OT peut effectuer un maximum de $\frac{n}{4}$ passes. Cependant, il existe aussi des arbres de fort diamètre pour lesquels OT effectue peu de passes (par l'exemple sur le graphe de la figure 3.6 page 78).

3.2.2 Etude du nombre de requêtes

Dans ce qui va suivre, de la même façon que pour les algorithmes LL, SLL et ASLL étudiés dans le chapitre 2, nous allons compter le nombre de récupérations de voisins effectuées par l'algorithme OT. Comme il peut effectuer plusieurs passes, nous nous intéresserons tout d'abord au nombre de requêtes effectuées lorsque le nombre de passes est minimal, c'est-à-dire égal à 1, puis nous étudierons sa complexité lorsque ce nombre est maximal.

Avant de s'intéresser au deux cas extrêmes évoqués ci-dessus, nous pouvons d'ores et déjà analyser le comportement global de OT. Soit u un sommet quelconque d'un arbre T. Durant l'exécution de l'algorithme OT, ce sommet u est marqué à la suite de plusieurs évènements.

1. Il s'agit d'une feuille.
2. Il s'agit d'un sommet isolé.
3. Il est le voisin d'une feuille (dans ce cas, il est ajouté à la solution).

On distingue la situation dans laquelle u est *directement supprimé* (cas 1 et 2) de celle dans laquelle il est *indirectement supprimé* (troisième cas). Dans la seconde situation, il peut très bien être marqué avant ou après son traitement.

Si l'exécution de l'algorithme OT nécessite plus d'une passe, le sommet u peut très bien être marqué entre la première et la dernière passe. S'il n'est pas supprimé directement lors d'une passe (c'est le cas si son degré courant est supérieur ou égal à 2), il peut faire l'objet d'au moins 2 requêtes et d'au plus $d(u)$ requêtes. Ce nombre de requêtes varie selon l'ordre dans lequel les voisins sont récupérés, le pire des cas étant de récupérer d'abord tous les voisins marqués.

Lorsque le nombre de passes est minimal

Dans ce qui va suivre, nous nous intéresserons à la complexité de l'algorithme OT lorsqu'il n'effectue qu'une seule passe sur l'arbre qu'il traite. En effet, comme nous l'avons énoncé dans le théorème 12 puis démontré, OT est toujours capable d'effectuer une seule passe et ce sur n'importe quel arbre.

Proposition 4. *Soit $T = (V, E)$ un arbre quelconque constitué de $\gamma + 1$ couches de sommets. Considérons l'ordre de récupération des sommets tel que* OT *n'effectue qu'une passe sur T et tel qu'il récupère pour chaque sommet leurs voisins déjà marqués en dernier. Notons O_*^* cet ordre. Soit $\mathfrak{q}_{\mathsf{OT}}(T, O_*^*)$ le nombre de requêtes effectuées sur l'arbre T avec l'ordre O_*^*. On a*

$$\mathfrak{q}_{\mathsf{OT}}(T, O_*^*) \quad \leq \sum_{u \in \Psi_i \ \wedge \ i \ pair} d(u) \ . \tag{3.1}$$

Démonstration. Soit $T = (V, E)$ un arbre constitué de $\gamma + 1$ couches de sommets. Soit u un sommet quelconque de T. On distingue deux cas.

1. Lorsque $u \in \Psi_k$, avec $k = 2q$ (k est pair) :
   - si u n'a pas déjà été marqué en tant que voisin d'un sommet, alors on compte au plus une fois son degré, ce qui fait exactement $d(u)$ requêtes.
2. Lorsque $u \in \Psi_k$, avec $k = 2q + 1$ (k est impair) :
   - ce sommet a déjà été marqué puisqu'il a été ajouté dans la solution, on ne compte donc pas de requête.

On obtient donc une borne supérieure sur le nombre minimum de requêtes effectuées par l'algorithme OT sur un arbre T. □

Par exemple, sur des chemins de taille impaire, dans le meilleur des cas, OT effectue m requêtes. Par contre, sur des chemins de taille paire, il en effectue « seulement » $m-1$. Sur des arbres binaires complets de hauteur d, notés ABC_d, avec un nombre de sommets $n = 2^{d+1} - 1$, OT effectue

$$\begin{aligned} \mathfrak{q}_{\mathsf{OT}}(ABC_d, O_*) &= 2^d + 3 \times 2^{d-2} + 3 \times 2^{d-4} + \cdots + 2^1 \\ &= \sum_{i=1}^{d} 2^i \ = \ 2^{d+1} - 2 \end{aligned}$$

requêtes, autrement dit m.

Remarquons que dans l'ordre de récupération O_*^*, dès que l'on traite un sommet, s'il n'est pas déjà marqué, il s'agit (dans l'arbre mis à jour) soit d'une feuille, soit d'un sommet isolé. Pour ces deux catégories de sommets, quel que soit l'ordre dans lequel leurs voisins sont scannés, on doit de toute façon effectuer $d(u)$ requêtes. On peut donc simplifier O_*^* en O_*.

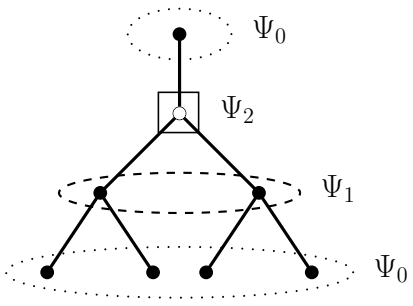


FIG. 3.5 – Arbre pour lequel l'algorithme OT effectue moins de m requêtes

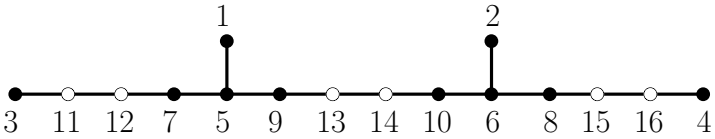


FIG. 3.6 – Peigne contenant 3 arêtes doublement couvertes

A la recherche d'une borne minimale plus précise. Dans la proposition 4, nous avons donné une borne supérieure sur le nombre minimum de requêtes effectuées par OT. On cherche maintenant à donner une borne minimale atteignable pour tout arbre. En effet, reprenons la preuve de la proposition 4. Les sommets u qui appartiennent aux couches paires ne font pas tous l'objet de $d(u)$ requêtes : certains d'entre eux peuvent avoir été ajoutés et marqués avant leur traitement, lorsqu'ils apparaissaient en tant que voisin d'un sommet d'une couche non voisine. Cette situation est possible par exemple sur des arbres pour lesquels un nombre maximum de $\left\lceil \frac{\gamma+1}{2} \right\rceil$ passes n'est pas atteignable. Sur l'arbre illustré par la figure 3.5, si l'algorithme OT scanne les sommets dans l'ordre des couches, le sommet en blanc est alors marqué avant son traitement. Il appartient pourtant à une couche paire.

De façon plus générale, OT va effectuer moins de m requêtes s'il retourne une solution qui contient des arêtes *doublement couvertes*, c'est-à-dire des arêtes dont les deux extrémités sont placées dans la solution. En effet, si elles sont scannées en dernier, elles ne sont pas considérées. La figure 3.6 montre un exemple d'arbre, que l'on appelera *peigne*, qui comporte trois arêtes doublement couvertes (il s'agit des arêtes dont les extrémités sont blanches). Si OT scanne les sommets dans l'ordre croissant des labels, ces arêtes ne seront pas scannées.

Généralisons l'arbre de la figure 3.6 et considérons un peigne à x branches reliant $x+1$ chemins P_4 entre eux. On a donc $m = 3(x+1) + 3x = 6x + 3$. Il y a une arête doublement couverte dans chacun des chemins P_4. Il y en a donc $x+1$ en tout. Par conséquent, OT effectue dans le meilleur des cas $\frac{6x+3}{x+1} \approx \frac{5m}{6}$ requêtes sur ce type d'arbre.

On peut donc donner une borne minimale sur le nombre de requêtes effectuées par l'algorithme OT pour tout arbre T. On a alors

$$\mathcal{B}\{\mathtt{q}_{\mathsf{OT}}(T)\} \;=\; m - \lambda \;,$$

avec λ le nombre d'arêtes doublement couvertes par OT.

Lorsque le nombre de passes est maximal

Nous nous intéressons à présent à la complexité en nombre de requêtes de l'algorithme OT lorsqu'il effectue un maximum de passes.

Lemme 6. *Soit $T = (V, E)$ un arbre quelconque constitué de $\gamma+1$ couches de sommets. Considérons l'ordre de récupération des sommets tel que* OT *effectue un maximum de passes sur T et tel qu'il récupère pour chaque sommet leurs voisins déjà marqués en dernier. Notons O_w^* cet ordre. Soit $\mathtt{q}_{\mathsf{OT}}(T, O_w^*)$ le nombre de requêtes effectuées sur l'arbre T avec l'ordre O_w^*. On a*

$$\mathtt{q}_{\mathsf{OT}}(T, O_w^*) \leq m + \sum_{i=0}^{\gamma} i \cdot \omega_i + \sum_{i \ impair} \omega_i \ . \tag{3.2}$$

Cette borne n'est pas atteinte pour tout arbre.

Démonstration. Soit $T = (V, E)$ un arbre constitué de $\gamma+1$ couches de sommets. Soit u un sommet quelconque de T. On distingue deux situations.

1. Lorsque $u \in \Psi_k$, avec $k = 2q$ (k est pair) :
   - pour les q passes durant lesquelles il n'est pas marqué, on effectue au plus deux requêtes à chaque fois, ce qui donne $2q = k$ requêtes en tout ;
   - lors de la $(q+1)^{\text{ème}}$ passe, si u n'a pas déjà été marqué en tant que voisin d'un sommet, alors on compte une fois son degré, ce qui donne $d(u)$ requêtes en plus.
2. Lorque $u \in \Psi_k$, avec $k = 2q+1$ (k est impair) :
   - durant les $q+1$ passes, on effectue au plus deux requêtes à chaque fois, ce qui donne $2q + 2 = k + 1$ requêtes en tout.

En effectuant la somme de ces différentes valeurs, on obtient donc

$$\begin{aligned} \mathtt{q}_{\mathsf{OT}}(T, O_w^*) &\leq \sum_{i \text{ pair}} i \cdot \omega_i + \sum_{u \in \Psi_i \ \wedge \ i \text{ pair}} d(u) + \sum_{i \text{ impair}} (i+1) \cdot \omega_i \\ &\leq m + \sum_{i=0}^{\gamma} i \cdot \omega_i + \sum_{i \text{ impair}} \omega_i \ . \end{aligned}$$

Cette valeur n'est pas atteinte sur les arbres pour lesquels OT effectue dans le pire des cas strictement moins de $\left\lceil \frac{\gamma+1}{2} \right\rceil$ passes (c'est le cas par exemple pour l'arbre de la figure 3.5). □

On peut maintenant donner une borne supérieure sur le nombre maximum de requêtes effectuées par l'algorithme OT pour tout arbre.

Le pire ordre de récupération possible des voisins d'un sommet est d'obtenir en premier tous ceux qui ont déjà été marqués par OT.

Théorème 14. *Soit $T = (V, E)$ un arbre quelconque constitué de $\gamma + 1$ couches de sommets. Soit $\mathcal{W}\{\mathfrak{q}_{\mathsf{OT}}(T)\}$ le nombre maximum de requêtes effectuées par l'algorithme OT exécuté sur l'arbre T. On obtient*

$$\mathcal{W}\{\mathfrak{q}_{\mathsf{OT}}(T)\} \leq \left\lceil \frac{\gamma+1}{2} \right\rceil \left(2n + 4 - 4 \left\lceil \frac{\gamma+1}{2} \right\rceil \right) - 2 \ . \tag{3.3}$$

Cette borne n'est pas atteinte pour tout arbre.

Démonstration. Pour atteindre cette borne, il faut que OT effectue un maximum de passes et qu'il marque un minimum de sommets lors de chaque passe, afin de maximiser le nombre de requêtes lors de la passe suivante. C'est le raisonnement que nous allons développer dans ce qui va suivre.

Soit $T = (V, E)$ un arbre constitué de $\gamma + 1$ couches de sommets. Lors de chaque passe, l'algorithme OT effectue un maximum de $d(u)$ requêtes pour chaque sommet u non encore marqué dans l'arbre (il ne peut pas faire pire). De plus, durant une passe, il marque au moins deux couches de sommets. Par conséquent, durant la $j^{\text{ème}}$ passe, il peut effectuer un maximum de

$$\sum_{u \in V(T_{2(j-1)})} d_T(u)$$

requêtes, avec $V(T_{2(j-1)})$ l'ensemble des sommets de l'arbre $T_{2(j-1)}$ qui résulte des suppressions des i couches de sommets Ψ_i, telles que $0 \leq i < 2(j-1)$, et avec $d_T(u)$ le degré du sommet u dans l'arbre T initial. On obtient alors un maximum de

$$\sum_{j=1}^{\lceil \frac{\gamma+1}{2} \rceil} 2 \cdot m(T_{2(j-1)})$$

requêtes, avec $m(T_{2(j-1)})$ le nombre d'arêtes de l'arbre $T_{2(j-1)}$.

On sait d'autre part que les couches d'un arbre sont constituées d'au moins deux sommets (sauf la dernière qui peut n'en contenir qu'un). De plus, ces sommets sont au minimum de degré 1 (ceux de la couche Ψ_0) ou de degré 2 (ceux des autres couches). Ainsi, lors de la première passe, OT marque au minimum deux sommets de degré 1 et deux sommets de degré 2. Ensuite, lors des passes suivantes, il marque au minimum quatre sommets de degré 2 (les feuilles « naturelles » de l'arbre ont disparu à la suite de la première passe). On obtient donc

$$\begin{aligned} \mathcal{W}\{\mathfrak{q}_{\mathsf{OT}}(T)\} &\leq 2m + 2(m-3) + \sum_{j=3}^{\lceil \frac{\gamma+1}{2} \rceil} 2(m - 3 - 4(j-2)) \\ &\leq \left\lceil \frac{\gamma+1}{2} \right\rceil \left(2m + 6 - 4 \left\lceil \frac{\gamma+1}{2} \right\rceil \right) - 2 \ . \end{aligned}$$

Si l'on remplace m par $n-1$, on obtient bien (3.3).

Cette borne est atteinte sur les chemins. En effet, dans les chemins, chaque couche de sommets est constituée de deux sommets (sauf éventuellement la dernière) qui sont de degré 1 (les feuilles

de la couche Ψ_0) ou bien de degré 2 (tous les autres sommets). De ce fait, dans le pire des cas, OT marque quatre sommets lors de chaque passe (sauf éventuellement à la dernière où il peut n'en marquer que trois ou un). De plus, pour tout i allant de 1 à $\gamma - 1$, le voisinage des sommets de la couche Ψ_i est réparti dans les couches Ψ_{i-1} et Ψ_{i+1}, et les voisins de la couche Ψ_0 (resp. Ψ_γ) sont tous dans Ψ_1 (resp. $\Psi_{\gamma-1}$ et éventuellement Ψ_γ). Par conséquent, sur un chemin à n sommets, le nombre de passes effectuées au maxium par l'algorithme OT est de $\lceil \frac{n}{4} \rceil$. On obtient donc

$$\begin{aligned} \mathcal{W}\{\mathtt{Q}_{\mathsf{OT}}(P_n)\} &= \left\lceil \frac{n}{4} \right\rceil \left(2n + 4 - 4\left\lceil \frac{n}{4} \right\rceil\right) - 2 \\ &= \left\lceil \frac{n^2}{4} + n - 2 \right\rceil \ . \end{aligned}$$

De manière générale, cette borne n'est pas atteinte sur des arbres pour lesquels OT ne peut effectuer un maximum de $\left\lceil \frac{\gamma+1}{2} \right\rceil$ passes. Elle n'est pas non plus atteinte lorsque l'arbre contient des sommets de degré strictement supérieur à 2. En effet, si aucun des voisins d'un sommet n'a été marqué (c'est souvent le cas lors de la première passe), quel que soit l'ordre dans lequel ils sont récupérés, on s'aperçoit seulement au bout de 2 requêtes que ce sommet n'est ni une feuille, ni un sommet isolé. □

Comme nous avons pu le voir dans la preuve du théorème 14, la complexité en nombre de requêtes sur les chemins, c'est-à-dire sur des arbres pour lesquels nous pensons que le nombre de passes est le plus fort, est en $\mathcal{O}(m^2)$.

A ce jour, nous n'avons pas trouvé d'arbres pour lesquels cette complexité était plus élevée.

3.3 Deux autres heuristiques adaptables : LR et ED

L'algorithme ED a été présenté dans la section 1.4 (voir page 29). L'algorithme LR est une version équivalente de l'algorithme *ListRight*, présenté lui aussi dans la section 1.4 (voir page 33).

Contrairement à OT, nous allons les implémenter de telle sorte qu'ils n'effectuent qu'une seule passe.

Dans ce qui va suivre, nous allons étudier les bornes minimales et maximales du nombre de requêtes qu'ils effectuent (des travaux relatifs à la qualité des solutions qu'ils construisent ont déjà été réalisés par *F. Delbot* [46]).

3.3.1 Adaptation de l'algorithme LR

Dans ce qui va suivre, nous allons nous baser sur une version alternative (mais équivalente) de l'algorithme *ListRight* décrit dans le chapitre 1. Il s'agit plus exactement de la version gloutonne (appelée *LRglouton*) décrite dans [46] et dans [27]. Nous avons choisi de nous baser sur cette version parce qu'elle est plus simple à mettre en œuvre et surtout, parce qu'elle est plus performante en terme de complexité en nombre de requêtes. Pour éviter toute confusion entre les deux versions, nous l'appelons LR.

Algorithme 12 (LR)**.** Soit $G = (V, E)$ un graphe. Soit C la couverture construite (au départ, $C = \emptyset$). Pour chaque sommet $u \in V$, si $u \notin C$, alors on ajoute tous ses voisins dans la solution ($C \leftarrow C \cup N(u)$).

Par rapport à la description générique fournie dans l'algorithme 12, vis-à-vis de notre modèle, on peut constater dans l'implémentation 2 que l'on envoie dans la solution uniquement les sommets qui n'y sont pas déjà. Ainsi, quelle que soit la forme sous laquelle la couverture est représentée (un ensemble, une liste, un flux de données, *etc.*), nous sommes certains de ne pas avoir de doublons et ce, sans avoir à effectuer d'opérations coûteuses supplémentaires. En effet, on se contente uniquement de tester si le drapeau matérialisant l'appartenance d'un sommet est levé ou non.

Implémentation 2 : LR

Données : un graphe étiqueté $G = (V, L, E)$ à n sommets
Résultat : une couverture $C \subseteq V$ de G

```
début
    Mem.init(0)
    pour i = 1 à n faire
        u ← getVertex(i)
        si Mem[L(u)] = 0 alors
            j ← 1
            tant que j ≤ d(u) faire
                /* On scanne l'arête uv.                                  */
                v ← getNeighbor(u, j)
                /* On met dans la solution les sommets qui n'y sont pas déjà.  */
                si Mem[L(v)] = 0 alors
                    v → C
                    Mem[L(v)] ← 1
                j ← j + 1
fin
```

Analyse de la complexité de LR en nombre de requêtes

Dans ce qui va suivre, nous nous baserons sur la version de LR décrite et détaillée dans l'implémentation 2.

Proposition 5. *Soit $G = (V, E)$ un graphe quelconque. Soit C_{LR} une couverture générée par l'algorithme* LR *pour le graphe G. On note $\mathtt{q}_{\mathsf{LR}}(G)$ le nombre de requêtes effectuées par* LR *exécuté sur G. On a*

$$\mathtt{q}_{\mathsf{LR}}(G) \;=\; \sum_{u \notin C_{\mathsf{LR}}} d(u) \;. \tag{3.4}$$

Démonstration. Soit $G = (V, E)$ un graphe. Soit u un sommet quelconque de G. De manière assez simple, lorsque u est examiné, s'il fait déjà partie de la solution, alors aucun de ses voisins n'est scanné. Autrement, lorsque u n'est pas encore dans la couverture, tous ses voisins sont examinés (ceux qui ne sont pas déjà dans la solution sont alors rajoutés). □

On remarque que $\mathcal{B}\{\mathtt{q}_{\mathsf{LR}}(G)\} = \mathcal{W}\{\mathtt{q}_{\mathsf{LR}}(G)\}$. La complexité de LR ne dépend donc finalement que des tailles des couvertures qu'il génère (l'ordre de récupération des voisins n'y change rien).

Aussi, quel que soit l'algorithme implémenté dans notre modèle, s'il retourne la même couverture que l'algorithme LR, il ne peut pas effectuer moins de requêtes que lui. En effet, il est contraint d'effectuer $d(u)$ requêtes pour tous les sommets qu'il ne met pas dans la couverture et, de manière triviale, il ne peut pas effectuer moins de zéro requête pour chaque sommet qu'il envoie dans la solution.

Théorème 15. *Soit $G = (V, E)$ un graphe quelconque. On note $\mathcal{W}\{\mathtt{q}_{\mathsf{LR}}(G)\}$ le nombre maximum de requêtes effectuées par l'algorithme* LR *exécuté sur le graphe G. On a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{LR}}(G)\} \leq m \ . \tag{3.5}$$

Cette borne n'est pas atteinte pour tout graphe.

Démonstration. Soit $G = (V, E)$ un graphe. Soit C_{LR} une couverture quelconque générée par l'algorithme LR pour le graphe G. Considérons une arête uv quelconque de G. Supposons sans perte de généralité que le sommet u apparaisse avant le sommet v durant l'exécution de LR. Si au moment de son examen, $u \in C_{\mathsf{LR}}$, alors on ne récupère aucun de ses voisins : l'arête uv n'est pas scannée (mais elle peut très bien l'être au moment de l'examen de v). Au contraire, si $u \notin C_{\mathsf{LR}}$, alors uv est scannée et v est mis dans la solution (s'il n'en fait pas déjà partie). De ce fait, par la suite, lorsque LR scanne v, il ne récupère pas ses voisins. Par conséquent, on peut dire que n'importe quelle arête uv appartenant à un graphe G quelconque ne peut pas être scannée deux fois. Donc, l'algorithme LR ne peut pas effectuer plus de m requêtes lors de son exécution.

Cette borne est atteinte sur les étoiles. Supposons que LR scanne le centre en premier. Il met alors tous ses voisins dans la solution. Il n'effectue plus de requête par la suite puisque tous les autres sommets de l'étoile sont dans la couverture.

Cette borne n'est pas atteinte sur les graphes complets K_n (avec $n > 2$). En effet, lorsque LR scanne le premier sommet, il ajoute tous les autres sommets du graphe dans la solution (ce qui nécessite $n - 1$ requêtes). Il n'effectue donc plus de requêtes par la suite. □

Le fait d'implémenter la version gloutonne de *ListRight* permet d'améliorer la complexité en nombre de requêtes. En effet, reprenons le raisonnement de la preuve de la proposition 5. Soit u un sommet scanné durant l'exécution de *ListRight* (la version décrite page 33). Lorsqu'on l'examine, on sait de manière certaine qu'il ne fait pas encore partie de la solution, puisque les sommets ne peuvent être ajoutés que lors de leur examen (ni avant, ni après). On distingue alors deux cas : celui où u va être mis dans la solution et celui où il ne va pas être rajouté.

1. Si $u \notin C_{\mathsf{LR}}$, alors, de la même façon que pour tous les autres algorithmes, il faut examiner tous ses voisins avant de le décider de manière définitive, ce qui génère $d(u)$ requêtes.
2. Si $u \in C_{\mathsf{LR}}$, alors cela signifie qu'il possède au moins un voisin qui n'est pas déjà dans la solution, ce qui génère dans le meilleur des cas 1 requête.

L'algorithme *ListRight* génère donc lors de son exécution sur un graphe G au moins $|C_{\mathsf{LR}}|$ requêtes supplémentaires par rapport à sa version gloutonne LR.

3.3.2 Adaptation de l'algorithme ED

Tel qu'il a été décrit précédemment (voir l'algorithme 2 page 29), cet algorithme traite le graphe sur lequel il est exécuté arête par arête, en les récupérant dans un ordre arbitraire.

Nous avons choisi de l'adapter de telle sorte à ce qu'il traite les graphes sommet par sommet et ce pour plusieurs raisons : d'une part, dans un souci de lisibilité vis-à-vis de la description des autres algorithmes que nous étudions ; d'autre part, pour obtenir (peut-être) un gain sur ses temps d'exécution. En effet, nous allons voir qu'il ne peut pas faire pire que m requêtes (ce qu'il ferait à chaque exécution s'il récupérait le graphe arête par arête) et qu'il peut même faire mieux.

Implémentation 3 : ED

Données : un graphe étiqueté $G = (V, L, E)$ à n sommets
Résultat : une couverture $C \subseteq V$ de G

```
début
    Mem.init(0)
    pour i = 1 à n faire
        u ← getVertex(i)
        si Mem[L(u)] = 0 alors
            j ← 1
            tant que j ≤ d(u) faire
                /* On scanne l'arête uv.                          */
                v ← getNeighbor(u, j)
                /* On recherche une arête non couverte.           */
                si Mem[L(v)] = 1 alors j ← j + 1
                sinon
                    u → C
                    v → C
                    Mem[L(u)] ← 1
                    Mem[L(v)] ← 1
                    j ← d(u) + 1
fin
```

Le principe global de l'algorithme ED est de choisir à chaque étape une arête non encore couverte et de placer dans la solution ses deux extrémités. Tel qu'il est décrit dans l'implémentation 3, il recherche une arête non encore couverte de la manière suivante. Pour chaque sommet u scanné, il teste dans un premier temps s'il fait partie de la solution (si oui, le bit mémoire relatif à ce sommet a pour valeur 1). Si c'est le cas, il ne regarde pas ses voisins (car toutes les arêtes $uv \mid v \in N(u)$ sont déjà couvertes) et il passe au sommet suivant. Autrement, il récupère ses voisins un par un, jusqu'à ce qu'il en trouve un qui ne fasse pas non plus partie de la solution (si tous les voisins de u sont dans la solution, on passe au sommet suivant et on applique le même principe, jusqu'à trouver un sommet avec un voisin qui ne font pas partie tous les deux de la solution). Une fois que l'algorithme a mis un sommet et l'un de ses voisins dans la solution, il passe au sommet suivant et recherche à nouveau une arête à couvrir.

Au final, on retourne bien les sommets d'un couplage maximal pour l'inclusion (et donc une couverture).

Analyse de la complexité de ED en nombre de requêtes

Dans ce qui va suivre, nous nous baserons sur la version de ED décrite et détaillée dans l'implémentation 3.

Proposition 6. *Soit $G = (V, E)$ un graphe quelconque. Soit C_{ED} une couverture générée par l'algorithme* ED *pour le graphe G. On note $\mathcal{B}\{\mathtt{q}_{\mathsf{ED}}(G)\}$ le nombre minimum de requêtes effectuées par l'algorithme* ED *exécuté sur le graphe G pour retourner la couverture C_{ED}. On a*

$$\mathcal{B}\{\mathtt{q}_{\mathsf{ED}}(G)\} \quad = \quad \sum_{u \notin C_{\mathsf{ED}}} d(u) + \frac{|C_{\mathsf{ED}}|}{2} \ . \tag{3.6}$$

Démonstration. Soit $G = (V, E)$ un graphe. Soit C_{ED} une couverture quelconque générée par l'algorithme ED pour le graphe G. On considère un sommet u de G. Pour que u ne fasse pas partie de la solution après avoir été examiné, il faut que tous ses voisins en fasse partie (autrement, le premier voisin v rencontré et qui n'est pas non plus dans la solution entraînera l'ajout de u et de v). Comme on ne sait pas déterminer à l'avance l'appartenance des voisins de u à la solution durant l'exécution de l'algorithme, il faut donc les récupérer et les tester, d'où les $d(u)$ requêtes.

A l'inverse, si le sommet u est ajouté à la couverture, dans le meilleur des cas, on trouve un voisin libre au bout d'une seule requête. On obtient donc $\frac{|C_{\mathsf{ED}}|}{2}$ requêtes pour l'ensemble des sommets, puisque à chaque fois que l'on ajoute un sommet u à la couverture, on ajoute aussi son voisin libre v. Or, on sait avec certitude que ce dernier n'a pas été examiné auparavant, sinon, u ne ferait pas l'objet de requêtes (il aurait fait partie des voisins déjà sélectionnés de v). On sait aussi de façon sûre qu'il ne sera pas examiné par la suite, puisqu'il vient d'être sélectionné. L'algorithme ED effectue donc dans le meilleur des cas $\frac{|C_{\mathsf{ED}}|}{2}$ requêtes pour les sommets qu'il ajoute à la solution. □

Théorème 16. *Soit $G = (V, E)$ un graphe quelconque. On note $\mathcal{W}\{\mathsf{q}_{\mathsf{ED}}(G)\}$ le nombre maximum de requêtes effectuées par l'algorithme* ED *exécuté sur le graphe G. On a*

$$\mathcal{W}\{\mathsf{q}_{\mathsf{ED}}(G)\} \quad \leq \quad m \ . \tag{3.7}$$

Cette borne n'est pas atteinte pour tout graphe.

Démonstration. Soit $G = (V, E)$ un graphe. Soit C_{ED} une couverture quelconque générée par l'algorithme ED pour le graphe G. Considérons une arête uv quelconque de G. Supposons sans perte de généralité que le sommet u apparaisse avant le sommet v durant l'exécution de ED. Si au moment de son examen, $u \in C_{\mathsf{ED}}$, alors on ne récupère aucun de ses voisins : l'arête uv n'est pas scannée (mais elle peut très bien l'être au moment de l'examen de v). Au contraire, si $u \notin C_{\mathsf{ED}}$, alors uv peut être scannée lors de l'examen de u. Supposons que ce soit le cas.

1. Si $v \in C_{\mathsf{ED}}$, alors lorsqu'il sera examiné par la suite, aucun de ses voisins ne sera récupéré, donc l'arête uv ne sera là aussi pas scannée une nouvelle fois.
2. Si $v \notin C_{\mathsf{ED}}$, alors u et v sont placés dans la solution et de ce fait, par la suite, lorsque le sommet v sera examiné, aucun de ses voisins ne sera récupéré, donc l'arête uv ne sera pas scannée une deuxième fois.

Par conséquent, on peut dire de manière générale que n'importe quelle arête uv appartenant à un graphe G quelconque ne peut pas être scannée deux fois. De ce fait, l'algorithme ED ne peut pas effectuer plus de m requêtes lors de son exécution.

Il existe des graphes pour lesquels cette borne est atteinte à chaque exécution. Considérons une étoile $S_n = (V, E)$ à n sommets et $n - 1$ arêtes. Sur ce graphe, l'algorithme ED scanne toutes les arêtes. En effet, après avoir analysé la première arête rencontrée (elles sont toutes de la forme cv_i, avec c le centre de S_n et v_i une feuille quelconque), quelle que soit la façon dont elle est scannée (si c est le sommet courant et v_i son voisin ou l'inverse), toutes les autres feuilles (qui ne font pas partie de la couverture) vont faire l'objet d'une requête. On effectue donc m requêtes en tout, et ce quel que soit l'ordre de traitement des sommets.

Montrons maintenant que cette borne n'est pas atteinte pour tout graphe. Considérons un graphe complet $K_4 = (V, E)$, avec $n = 4$ et $m = 6$. La pire exécution possible en terme de complexité en nombre de requêtes examine 4 arêtes, dans l'ordre arbitraire suivant (que l'on peut considérer sans perte de généralité) : v_1v_2, v_3v_1, v_3v_2 et v_3v_4. A ce stade de l'exécution, la couverture est constituée des sommets v_1, v_2, v_3 et v_4. Autrement dit, elle contient déjà tous les sommets du graphe. Les arêtes restantes (v_2v_4 et v_4v_1) ne sont donc pas examinées. □

3.4 L'algorithme Pitt

Il s'agit d'un algorithme probabiliste assez simple décrit par *L. Pitt* [106]. Cet algorithme est en moyenne 2-approché. Il peut aussi facilement s'adapter au cas pondéré. De la même façon que pour LR et ED, après avoir donné une description détaillée de son implémentation dans notre

modèle, nous donnerons une borne inférieure et une borne supérieure de sa complexité en nombre de requêtes.

3.4.1 Présentation

Algorithme 13 (Pitt)**.** Soit $G = (V, E)$ un graphe. Soit C la couverture construite (au départ, $C = \emptyset$). Tant qu'il reste au moins une arête non couverte $uv \in E \mid \{u, v\} \cap C = \emptyset$, on joue à « Pile ou Face » (on jette une pièce équilibrée) : si le résultat est Pile, on ajoute u à C, sinon, on ajoute v.

La figure 3.7 montre un exemple d'exécution de l'algorithme Pitt sur un graphe.

Pour les mêmes raisons que pour l'algorithme ED, nous avons choisi d'adapter cet algorithme dans notre modèle de telle sorte à ce qu'il traite les graphes sommet par sommet. Bien qu'il existe des situations pour lesquelles Pitt peut effectuer $2m$ requêtes, il peut aussi avoir les mêmes performances que l'algorithme LR en terme de complexité en nombre de requêtes. Aussi, nous avons constaté que bien souvent, il effectuait moins de m requêtes (les cas où il effectue $2m$ requêtes sont assez rares).

De la même façon que pour ED, l'algorithme Pitt recherche une arête uv à couvrir en parcourant le graphe par paquets d'étoiles, c'est-à-dire sommet par sommet, en récupérant pour chaque sommet u ses voisins v un par un. S'il place le sommet u dans la solution, il passe au sommet suivant (car il a couvert toutes les arêtes $uv \mid v \in N(u)$). S'il place le voisin v dans la solution, il regarde les voisins restants de u (car certains peuvent ne pas faire partie de la solution).

Au final, on couvre bien toutes les arêtes du graphe.

Analyse de la complexité de Pitt en nombre de requêtes

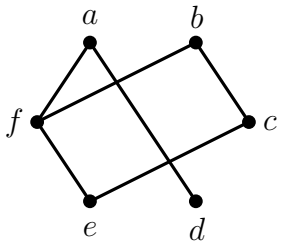
Proposition 7. *Soit $G = (V, E)$ un graphe quelconque. Soit C_{Pitt} une couverture générée par l'algorithme* Pitt *pour le graphe G. On note $\mathcal{B}\{\mathtt{q}_{\mathsf{Pitt}}(G)\}$ le nombre minimum de requêtes effectuées par* Pitt *exécuté sur le graphe G pour retourner la couverture C_{Pitt}. On a*

$$\mathcal{B}\{\mathtt{q}_{\mathsf{Pitt}}(G)\} \quad = \quad \sum_{u \notin C_{\mathsf{Pitt}}} d(u) \ . \tag{3.8}$$

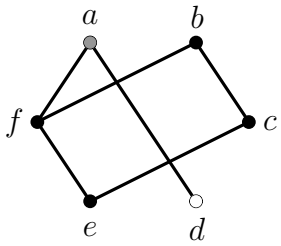
Démonstration. Soit $G = (V, E)$ un graphe. Soit C_{Pitt} une couverture quelconque produite par l'algorithme Pitt pour G. Il existe une exécution de Pitt sur G telle que tous les sommets $u \in C_{\mathsf{Pitt}}$ sont ajoutés dans la solution avant d'être examinés. Ces sommets sont ajoutés en tant que voisins d'autres sommets. On obtient une telle exécution de la façon suivante. Considérons un sommet quelconque u de G. Lors de son examen, si $u \notin C_{\mathsf{Pitt}}$, alors, pour chacun de ses voisins v, la pièce retombe sur Face et fait mettre v dans la couverture.

En fait, cela revient à exécuter l'algorithme LR sur G puisque, de cette façon, pour chaque sommet $u \notin C_{\mathsf{Pitt}}$, on met à chaque fois tous ses voisins dans la solution.

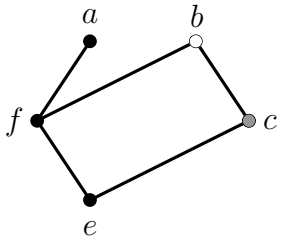
Cette exécution permet de minimiser le nombre de requêtes. En effet, de façon générale, on ne peut pas faire mieux que $d(u)$ requêtes pour chaque sommet $u \notin C_{\mathsf{Pitt}}$, puisque si u n'y est pas, alors tous ses voisins y sont (et on est donc obligé de tous les scanner). Aussi, pour chaque sommet



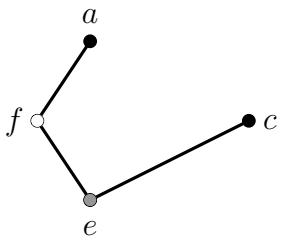
(a) Graphe sur lequel on exécute Pitt



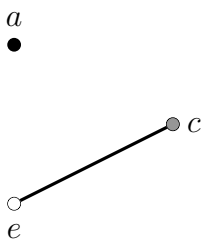
(b) Première étape : On joue à « Pile ou Face » avec l'arête ad : le sommet d (en blanc) est mis dans la solution



(c) Deuxième étape : on procède de la même façon avec l'arête fb

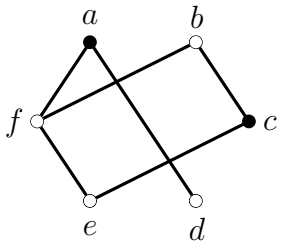


(d) Troisième étape : on procède de la même façon avec l'arête fe

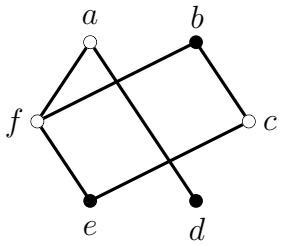


(e) Quatrième étape : on procède de la même façon avec l'arête ec

(f) Cinquième étape : il n'y a plus rien à faire (le graphe ne contient plus d'arête)



(g) Solution obtenue à la fin de l'exécution



(h) Solution optimale sur le graphe de (a)

FIG. 3.7 – Exécution de l'algorithme Pitt sur un graphe à 6 sommets

Implémentation 4 : Pitt

Données : un graphe étiqueté $G = (V, L, E)$ à n sommets
Résultat : une couverture $C \subseteq V$ de G

```
début
    Mem.init(0)
    pour i = 1 à n faire
        u ← getVertex(i)
        si Mem[L(u)] = 0 alors
            j ← 1
            tant que j ≤ d(u) faire
                /* On scanne l'arête uv.                                  */
                v ← getNeighbor(u, j)
                /* On recherche une arête non couverte.                   */
                si Mem[L(v)] = 1 alors j ← j + 1
                sinon
                    JetPiece ← « Pile ou Face »
                    si JetPiece = Pile alors
                        u → C
                        Mem[L(u)] ← 1
                        j ← d(u) + 1
                    sinon
                        v → C
                        Mem[L(v)] ← 1
                        /* On continue à explorer les voisins restants de u, pour
                           trouver d'autres arêtes non couvertes.          */
                        j ← j + 1
fin
```

$u \in C_{\mathsf{Pitt}}$, pour que l'on ne scanne pas leurs voisins, ils doivent tous avoir été rajoutés avant leur examen en tant que voisins d'autres sommets.

Le nombre minimum de requêtes effectuées par l'algorithme Pitt s'obtient donc lorsque, durant son exécution, il a le même comportement que l'algorithme LR, c'est-à-dire en mettant les sommets dans la solution uniquement lorsqu'ils sont scannés en tant que voisin. □

Théorème 17. *Soit $G = (V, E)$ un graphe quelconque. On note $\mathcal{W}\{\mathtt{q}_{\mathsf{Pitt}}(G)\}$ le nombre maximum de requêtes effectuées par* Pitt *sur le graphe G. On a*

$$\mathcal{W}\{\mathtt{q}_{\mathsf{Pitt}}(G)\} \leq 2m \ . \tag{3.9}$$

Cette borne n'est pas atteinte pour tout graphe.

Démonstration. Soit uv une arête quelconque d'un graphe. Elle ne peut pas être examinée plus de deux fois : elle l'est au moment de l'examen du sommet u et de son voisin v et au moment de l'examen du sommet v et de son voisin u. En effet, de la manière dont nous traitons les graphes dans notre modèle, nous ne pouvons pas faire pire que $2m$ requêtes puisque, dans une liste d'adjacence, cela revient à parcourir tous les voisins de tous les sommets une fois (car $\sum_{u \in V} d(u) = 2m$). Comme nous n'effectuons qu'une seule passe sur le graphe en entrée, $2m$ requêtes est donc la valeur maximale que l'on puisse atteindre.

Cette borne peut être atteinte sur les étoiles. Soit $S_n = (V, E)$ une étoile. Supposons que l'algorithme Pitt traite le centre en dernier. Il peut effectuer $2m$ requêtes de la manière suivante. Pour chaque feuille examinée, lorsqu'il joue à « Pile ou Face » avec son voisin (le centre de l'étoile), il met la feuille dans la solution. Cela génère exactement m requêtes. Lorsqu'il traite le centre de l'étoile en dernier, il doit récupérer tous ses voisins un par un pour finalement décider que toutes ses arêtes incidentes sont déjà couvertes. Cela coûte m requêtes supplémentaires. On a donc bien $2m$ requêtes au total.

Fort heureusement, il existe un certain nombre de graphes pour lesquels cette borne n'est pas atteinte. Considérons un graphe G quelconque dans lequel $\delta \geq 2$. Prenons par exemple un cycle $C_3 = (V, E)$, constitué des trois sommets u, v et w et des arêtes uv, uw et vw. Lorsque l'on examine un sommet et l'un de ses voisins, si l'on place systématiquement le sommet dans la solution, on aura au final une arête qui n'a pas été scannée deux fois puisque, lors de la première étape, quand u (le sommet) et v (le voisin) sont examinés, dès lors que u est placé dans la couverture, on ne regarde pas son voisin suivant (w) et on passe directement au sommet v. L'arête uw n'est donc pas scannée à ce moment là. Elle ne sera scannée qu'une seule fois, au moment de l'examen du sommet w et de son voisin u. □

L'algorithme Pitt a la particularité d'avoir une borne inférieure minimale (on ne peut pas faire mieux) et une borne supérieure maximale (on ne peut pas faire pire) sur le nombre de requêtes. Pitt a donc presque les mêmes caractéristiques pour le nombre de requêtes que LL pour la qualité des solutions construites. La seule différence réside dans le fait que pour Pitt, la borne maximale n'est pas atteinte pour tout graphe.

3.4.2 Restrictions sur la mémoire disponible

Pour aller plus loin dans l'étude de l'algorithme Pitt, nous nous sommes intéressés à la mémoire qu'il utilisait pour fonctionner. Nous décrivons pour cela une version de l'algorithme Pitt dans laquelle la taille mémoire maximale est donnée en paramètre.

Soit $\mathfrak{M}$ la mémoire disponible sur l'unité de traitement. Soit k sa taille, telle que $0 \leq k \leq n$. Dans les deux algorithmes que nous allons décrire, nous utiliserons $\mathfrak{M}$ comme un ensemble.

Par rapport aux algorithmes décrits précédemment et par rapport au modèle que nous avons décrit au début de ce chapitre, nous supposons cette fois-ci que la mémoire disponible sur l'unité de traitement est donc capable dans le pire des cas de contenir les n sommets du graphe, ce qui représente $\mathcal{O}(n \log n)$ bits. Notre objectif est d'utiliser de manière plus efficace cette mémoire, c'est-à-dire de faire mieux en terme d'espace utilisé que n bits. L'idéal serait donc de montrer qu'avec une mémoire de taille $k \leq \frac{n}{\log_2 n}$, l'algorithme Pitt est capable de retourner des solutions « presque aussi bonnes » qu'avec une mémoire de taille n.

Algorithme 14 (Pitt avec mémoire limitée)**.** Soit $G = (V, E)$ un graphe. Soit $\mathfrak{M}$ l'espace mémoire disponible, de taille k. Les arêtes sont traitées dans un ordre arbitraire. Pour chaque arête $uv \in E$ telle que $\{u, v\} \cap \mathfrak{M} = \emptyset$, on joue à « Pile ou Face » : si le résultat est Pile (resp. Face), on met u (resp. v) dans la solution et on l'ajoute à la mémoire, si elle n'est pas pleine (si $|\mathfrak{M}| < k$).

Afin d'étudier de manière plus classique l'algorithme Pitt, nous ne considérerons pas dans ce qui va suivre le modèle de traitement présenté à la section 1.2. Nous nous baserons donc directement sur la description générique de l'algorithme Pitt à mémoire limitée fournie par l'algorithme 14. Cela va nous permettre de simplifier les preuves présentées.

Un premier résultat que l'on peut énoncer dans le lemme 7 concerne la taille moyenne des couvertures qu'il construit lorsqu'il ne dispose d'aucune mémoire.

Lemme 7. *Soit $G = (V, E)$ un graphe quelconque. Soit $\mathbb{E}\big[\mathsf{Pitt}(G, 0)\big]$ la taille moyenne des solutions construites par l'algorithme* Pitt *pour le graphe G, lorsqu'il ne dispose d'aucune mémoire. En considérant tous les ordres de traitement des arêtes et tous les lancers de pièce de manière équiprobable, on a*

$$\mathbb{E}\big[\mathsf{Pitt}(G, 0)\big] \quad = \quad n - \sum_{u \in V} \frac{1}{2^{d(u)}} \ . \tag{3.10}$$

Démonstration. L'algorithme considère chaque arête indépendamment des autres. Ainsi, chaque sommet u du graphe est impliqué dans exactement $d(u)$ lancers de pièce, et n'est pas sélectionné avec la probabilité $\frac{1}{2}$ dans chacun de ces lancers. Par conséquent, la probabilité qu'un sommet u soit sélectionné par Pitt est égale à $1 - \frac{1}{2^{d(u)}}$. □

Sur le même schéma de comparaison, lorsqu'il n'y a pas de mémoire disponible sur l'unité de traitement, l'algorithme Pitt est donc plus mauvais en moyenne que l'algorithme LL.

Avec une mémoire de taille $k = \frac{n}{2}$, il existe des graphes pour lesquels l'algorithme Pitt peut être très mauvais en moyenne.

En effet, considérons un graphe quelconque à n sommets. Soit k la taille de l'espace mémoire disponible, telle que $k \leq \frac{n}{2}$. Soit OPT la taille d'une couverture optimale pour le graphe G. Supposons dans un premier temps que $OPT > k$. Lors de l'exécution de Pitt, on peut observer deux phases.

1. Tant que la mémoire n'est pas encore pleine, à chaque étape « utile », c'est-à-dire à chaque fois qu'il traite une arête non encore couverte, Pitt rajoute un nouveau sommet dans la couverture.
2. Une fois que la mémoire est remplie, Pitt travaille « en aveugle » sur les arêtes qu'il reste à couvrir (il y en a forcément, puisque par hypothèse, $OPT > k$). Si le graphe induit par $G - \mathfrak{M}$ a un degré minimum de 3, alors, d'après le lemme 7, on retourne en moyenne $\frac{7}{8}$ des sommets de $V \setminus \mathfrak{M}$.

Maintenant, considérons un graphe biparti complet $K_{k,x \cdot k} = (X \cup Y, E)$, avec k la taille de l'espace mémoire disponible et $x > 0$. On a bien $k \leq \frac{n}{2}$. Si l'on exécute l'algorithme Pitt sur ce graphe et avec ces paramètres, lors des k premières étapes, donc tant que la mémoire n'est pas pleine, on place environ $\frac{k}{2}$ sommets de X et $\frac{k}{2}$ sommets de Y en mémoire (et dans la couverture), puisque chaque arête possède une extrémité dans X et une extrémité dans Y. Une fois que la mémoire est pleine, l'exécution de l'algorithme se poursuit sur le graphe biparti complet restant $K'_{\frac{k}{2},(x-1)\cdot k+\frac{k}{2}} = (X' \cup Y', E')$ pour lequel on ne dispose plus de mémoire. Si l'on considère une valeur de k suffisamment grande pour que $\frac{k}{2} > 2$ (au moins 6), alors on obtient en moyenne une couverture au moins de taille $k + \frac{7}{8} \cdot kx$, et comme $n = k + x \cdot k$, on a un graphe sur lequel l'algorithme Pitt, exécuté avec une mémoire de taille $k \leq \frac{n}{2}$, retourne en moyenne plus de $\frac{7}{8}$ des sommets (pour $k = 6$). Ce ratio augmente lorsque k augmente.

La figure 3.8 montre un exemple d'un tel graphe. Il s'agit d'un graphe biparti complet $K_{6,12} = (X \cup Y, E)$. Tant que la mémoire n'est pas pleine, Pitt retourne en moyenne 3 sommets de X et 6 sommets de Y (il s'agit des sommets en blanc sur la figure 3.8). Et lorsque la mémoire est pleine, il retourne la plupart des sommets restants.

3.4.3 Vers une gestion plus fine de la mémoire : l'algorithme S-Pitt

A défaut de montrer que l'algorithme Pitt, exécuté avec une mémoire de taille $\frac{n}{\log_2 n}$ au maximum, a un bon comportement, nous avons mis au point une nouvelle version dans laquelle on profite d'une hypothèse supplémentaire pour réduire le nombre de tests effectués sur la mémoire. Il s'agit de l'algorithme S-Pitt.

Dans ce qui va suivre, nous supposerons que les sommets sont stockés dans l'ordre croissant des labels, c'est-à-dire de 1 jusqu'à n (par contre, les voisins des sommets sont toujours stockés dans un ordre quelconque). On dit alors que les arêtes apparaissent dans un ordre *semi-trié*. Le principe général de l'algorithme S-Pitt (détaillé dans l'implémentation 5) utilise fortement cette hypothèse supplémentaire pour gérer la mémoire.

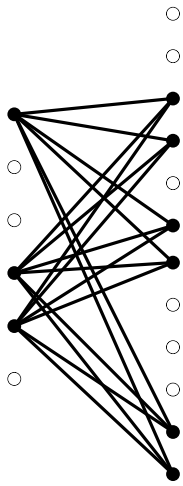


FIG. 3.8 – Graphe biparti complet pour lequel l'algorithme Pitt, avec une mémoire de taille $\frac{n}{2}$, retourne en moyenne quasiment tous les sommets

- Pour chaque sommet u, on ne considère que ses voisins droits.
- Si le voisin de u est ajouté à la couverture, alors on le met dans la mémoire.
- Si c'est le sommet u qui est mis dans la solution, alors on n'a pas besoin de le placer en mémoire. En effet, comme les sommets sont scannés par ordre croissant des labels et comme on ne regarde que les arêtes uv telles que $L(v) > L(u)$, on sait par avance que l'on ne rencontrera plus u par la suite.
- Enfin, si le sommet courant est dans la mémoire, on peut le retirer, car, pour les mêmes raisons que précédemment, il ne sera plus rencontré par la suite.

Grâce au fait que les arêtes soient semi-triées, même si la taille mémoire k donnée en paramètre est égale à 0, l'algorithme S-Pitt dispose tout de même d'une certaine façon de mémoire. Cela le rend plus efficace mais rend aussi son analyse et sa comparaison avec l'algorithme Pitt (au moins en terme de qualité de solution) plus complexe.

3.4.4 Comparaison des algorithmes Pitt et S-Pitt

Nous avons tenté d'établir une relation entre les deux algorithmes, par rapport à la qualité moyenne des solutions qu'ils retournent et par rapport à la quantité de mémoire qu'ils utilisent. Nous allons voir que l'algorithme S-Pitt n'est pas toujours meilleur en moyenne que l'algorithme Pitt.

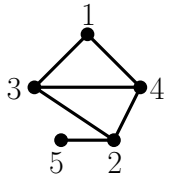
Etude sur les étoiles

Nous avons comparé de façon analytique les algorithmes Pitt et S-Pitt sur une famille de graphes simple : les étoiles. Nous sommes parvenus à déterminer les tailles moyennes des solutions qu'ils retournaient sur ces graphes.

Implémentation 5 : S-Pitt

Données : un graphe étiqueté $G = (V, L, E)$ à n sommets
Entrées : une taille mémoire k telle que $0 \leq k \leq n$
Résultat : une couverture $C \subseteq V$ de G

```
début
    𝔐 ← ∅
    pour i = 1 à n faire
        /* On récupère les sommets dans l'ordre croissant des labels.        */
        u ← getVertex(i)
        si u ∉ 𝔐 alors
            j ← 1
            tant que j ≤ d(u) faire
                /* On scanne l'arête uv.                                     */
                v ← getNeighbor(u, j)
                si L(u) < L(v) et v ∉ 𝔐 alors
                    JetPiece ← « Pile ou Face »
                    si JetPiece = Pile alors
                        u → C
                        j ← d(u) + 1
                    sinon
                        v → C
                        si |𝔐| < k alors 𝔐 ← 𝔐 ∪ {v}
                        /* On continue à explorer les voisins restants de u, pour
                           trouver d'autres arêtes non couvertes.             */
                        j ← j + 1
                sinon j ← j + 1
        sinon 𝔐 ← 𝔐 \ {u}
fin
```



(a) Graphe sur lequel on exécute l'algorithme S-Pitt

$1 \to 4, 3$
$2 \to 5, 4, 3$
$3 \to 2, 4, 1$
$4 \to 1, 3, 2$
$5 \to 2$

(b) Liste d'adjacence du graphe, parcourue de haut en bas et de gauche à droite par S-Pitt (ordre *semi-trié*)

i	j	$JetPiece$	$\mathfrak{M}$
1	2	Face	$\{4\}$
$C = \{4\}$			

(c) Première étape : état de la mémoire après le traitement de l'arête $\{1, 4\}$

i	j	$JetPiece$	$\mathfrak{M}$ (**pleine**)
2	3	Face	$\{4\}$
$C = \{4, 3\}$			

(d) Deuxième étape : état de la mémoire après le traitement de l'arête $\{1, 3\}$

i	j	$JetPiece$	$\mathfrak{M}$
3	4	Pile	$\{4\}$
$C = \{4, 3, 2\}$			

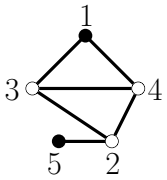
(e) Troisième étape : état de la mémoire après le traitement de l'arête $\{2, 5\}$

i	j	$JetPiece$	$\mathfrak{M}$
4	4	Pile	$\{4\}$
$C = \{4, 3, 2\}$			

(f) Quatrième étape : état de la mémoire après le traitement du sommet 3

i	j	$JetPiece$	$\mathfrak{M}$
5	2	Pile	$\emptyset$
$C = \{4, 3, 2\}$			

(g) Cinquième étape : état de la mémoire après le traitement des sommets 4 et 5



(h) Couverture obtenue à la fin de l'exécution

FIG. 3.9 – Exécution de l'algorithme S-Pitt (détaillé par l'implémentation 5) sur un graphe à 5 sommets, avec $k = 1$

Théorème 18. *Soit $S_n = (V, E)$ une étoile quelconque à n sommets. Soit k la taille mémoire disponible, tel que $1 \leq k \leq n$. On désigne par $\mathbb{E}\big[\mathsf{Pitt}(S_n, k)\big]$ (resp. $\mathbb{E}\big[\mathsf{S\text{-}Pitt}(S_n, k)\big]$) la taille moyenne des solutions retournées par l'algorithme* Pitt *(resp.* S-Pitt*) sur l'étoile S_n, lorsqu'il dispose d'un espace mémoire de taille k. En considérant tous les ordres de traitement des arêtes et tous les lancers de pièce de manière équiprobable, pour l'algorithme* Pitt*, on a*

$$\begin{aligned} \mathbb{E}\big[\mathsf{Pitt}(S_n, k)\big] &= \sum_{i=1}^{k} \frac{i}{2^i} + \frac{1}{2^k}\left(\frac{n-k}{2} + \frac{1}{2} - \frac{1}{2^{n-k-1}}\right) \\ &= 2 - \frac{k+2}{2^k} + \frac{1}{2^k}\left(\frac{n-k}{2} + \frac{1}{2} - \frac{1}{2^{n-k-1}}\right) \ . \end{aligned} \tag{3.11}$$

Et pour l'algorithme S-Pitt*, on obtient*

$$\begin{aligned} \mathbb{E}\big[\mathsf{S\text{-}Pitt}(S_n, k)\big] &= \sum_{i=1}^{n-1} \frac{i}{2^i} \\ &= 2 - \frac{n+1}{2^{n-1}} \ . \end{aligned} \tag{3.12}$$

Démonstration. Soit $S_n = (V, E)$ une étoile à n sommets. Soit k la taille de l'espace mémoire disponible sur la machine de traitement. Toutes les arêtes de S_n sont de la forme cv, avec c le centre de l'étoile et v une feuille quelconque.

On donne tout d'abord la preuve pour l'algorithme Pitt. Si le centre c est mis dans la solution avant k étapes, il est placé dans la mémoire. Par conséquent, toutes les arêtes scannées après lui seront déjà couvertes. Dans ce cas, le nombre moyen de sommets placés en mémoire avant c est égal à $\sum_{i=1}^{k} \frac{i}{2^i}$. En effet, le centre c entre dans la solution à la première étape avec une probabilité de $\frac{1}{2}$ (on a alors placé un sommet dans la couverture) ; le centre c entre dans la solution à la deuxième étape avec une probabilité de $\frac{1}{4}$ (on a alors placé deux sommets dans la couverture) ; *etc.* On obtient donc

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{i}{2^i} + \cdots + \frac{k}{2^k} \ ,$$

ce qui donne bien $\sum_{i=1}^{k} \frac{i}{2^i}$.

Les rares fois où le centre c n'est pas mis dans la mémoire au bout de k étapes (cela se produit avec une probabilité de $\frac{1}{2^k}$), la solution retournée par Pitt est très mauvaise, puisqu'il fonctionne « en aveugle » sur les arêtes restantes (qui ne sont pas encore couvertes). Cela revient donc à exécuter l'algorithme Pitt sans mémoire sur une étoile S_{n-k}. On applique donc (3.10) sur les $n-k$ sommets restants. On obtient donc

$$n - k - \frac{n-1-k}{2} - \frac{1}{2^{n-1-k}} \ ,$$

ce qui donne bien $\frac{n-k}{2} + \frac{1}{2} - \frac{1}{2^{n-k-1}}$, que l'on pondère avec la probabilité $\frac{1}{2^k}$ et que l'on regroupe avec le terme $\sum_{i=1}^{k} \frac{i}{2^i}$ pour obtenir (3.11).

Donnons à présent la preuve pour l'algorithme S-Pitt. Il scanne les sommets u (qui apparaissent par ordre croissant des labels) et leurs voisins v (qui apparaissent dans un ordre quelconque) un

par un, en ne considérant que les arêtes uv telles que v est voisin droit de u. Par conséquent, tant que le centre de l'étoile c n'est pas récupéré en tant que sommet, toutes les arêtes uv scannées sont considérées (puisqu'à chaque fois, u est une feuille et v le centre c, qui apparaît donc en tant que voisin droit). Après avoir scanné le centre (ainsi que ses voisins), l'algorithme S-Pitt n'ajoute plus de sommet dans la solution, puisqu'il ne reste plus d'arête à considérer. Aussi, du fait qu'il retire de sa mémoire les sommets qu'il vient de scanner, S-Pitt n'a besoin que d'un espace mémoire de taille 1 pour s'exécuter sur les étoiles, afin de stocker le centre lorsqu'il est placé dans la couverture avant d'être récupéré en tant que sommet. Il fonctionne donc comme s'il disposait de mémoire tout au long de son exécution. On applique alors le raisonnement établi précédemment avec l'algorithme Pitt pour prouver le terme $\sum_{i=1}^{k} \frac{i}{2^i}$. On remplace k par $n-1$ (il s'agit du nombre d'arêtes considérées par S-Pitt) et on obtient (3.12). □

L'algorithme S-Pitt est donc meilleur que l'algorithme Pitt sur les étoiles.

Le corollaire 6 donne des bornes sur la quantité de mémoire nécessaire à l'algorithme Pitt pour « rester efficace » (c'est-à-dire avoir les mêmes résultats en moyenne que l'algorithme qui utilise une mémoire de taille n) sur les étoiles.

Corollaire 6. *Soit z une constante. Pour toute étoile S_n à n sommets, si l'algorithme* Pitt *dispose d'un espace mémoire de taille $k = z \cdot \log_2 n$, alors*

- *si $z > 1$, il est 2-approché en moyenne ;*
- *si $z = 1$, il est $\frac{5}{2}$-approché en moyenne ;*
- *si $z < 1$, son rapport d'approximation en moyenne n'est plus constant.*

Démonstration. D'après le théorème 18, l'algorithme Pitt retourne en moyenne

$$\frac{2^{k+1} - k - 2}{2^k} + \frac{1}{2^k}\left(\frac{n-k}{2} + \frac{1}{2} - \frac{1}{2^{n-k-1}}\right)$$

sommets sur les étoiles S_n. Si l'on remplace k par $\log_2 n$, on obtient

$$\frac{5n - 3 - 3\log_2 n}{2n} - \frac{1}{2^{n-1}} \ ,$$

valeur qui tend vers $\frac{5}{2}$ lorsque n tend vers $+\infty$. De plus, comme on peut le voir sur la courbe 3D de la figure 3.10, ce rapport d'approximation moyen diminue lorsque la constante z augmente. □

Comparaison expérimentale sur des chemins

Nous avons choisi de comparer de manière expérimentale les algorithmes Pitt et S-Pitt sur des chemins de taille raisonnable (une centaine de sommets). Dans ce qui va suivre, nous exprimerons la quantité d'espace mémoire disponible par un coefficient appliqué au nombre de sommets n.

Les courbes de la figure 3.11 montrent le résultat d'expérimentations réalisées sur des chemins de taille $n = 100$. Soit α le coefficient appliqué à n pour exprimer la quantité de mémoire disponible.

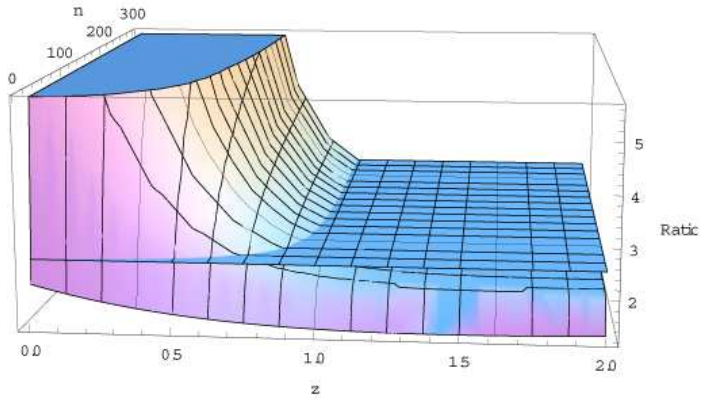


FIG. 3.10 – Courbe 3D montrant le rapport d'approximation moyen de l'algorithme Pitt à mémoire limitée obtenu sur les étoiles S_n, avec une mémoire de taille $k = z \cdot \log_2 n$ et avec $n \in [2, 300]$ et $z \in [0, 2]$

Pour chaque valeur de α, qui varie de 0 à 1 par pas de $\frac{1}{100}$, on exécute 3000 fois chaque algorithme. Les arêtes sont traitées à chaque fois dans un ordre arbitraire et les sommets sont étiquetés de façon aléatoire. Pour chaque valeur de α, on affiche la moyenne du rapport d'approximation moyen obtenu (sur un chemin de taille $n = 100$, la couverture optimale contient 50 sommets).

Comme on peut le voir sur les courbes de la figure 3.11, les deux courbes se croisent pour $\alpha \approx 0{,}52$. Avant cette intersection, l'algorithme S-Pitt est meilleur en moyenne que Pitt, puisque, tel qu'il est implémenté, à quantité de mémoire égale, il utilise en réalité plus de mémoire que Pitt.

Après cette intersection, l'algorithme Pitt devient meilleur en moyenne que l'algorithme S-Pitt. Ce phénomène peut s'expliquer de la manière suivante. S-Pitt traite les arêtes dans un ordre semi-trié : les sommets apparaissent par ordre croissant des labels. De plus, comme le graphe est récupéré sommet par sommet, les arêtes apparaissent par paquets d'étoiles (on ne peut pas avoir par exemple l'ordre de traitement suivant : uv, ab puis uw). L'algorithme S-Pitt a donc une « marge de manœuvre » plus restreinte que l'algorithme Pitt qui, tel qu'il est décrit dans l'algorithme 14, récupère les arêtes dans un ordre arbitraire.

Sur des chemins à 100 sommets, lorsque l'unité de traitement peut stocker en mémoire plus de la moitié des sommets, nous avons donc constaté expérimentalement que l'algorithme Pitt était meilleur en moyenne que l'algorithme S-Pitt.

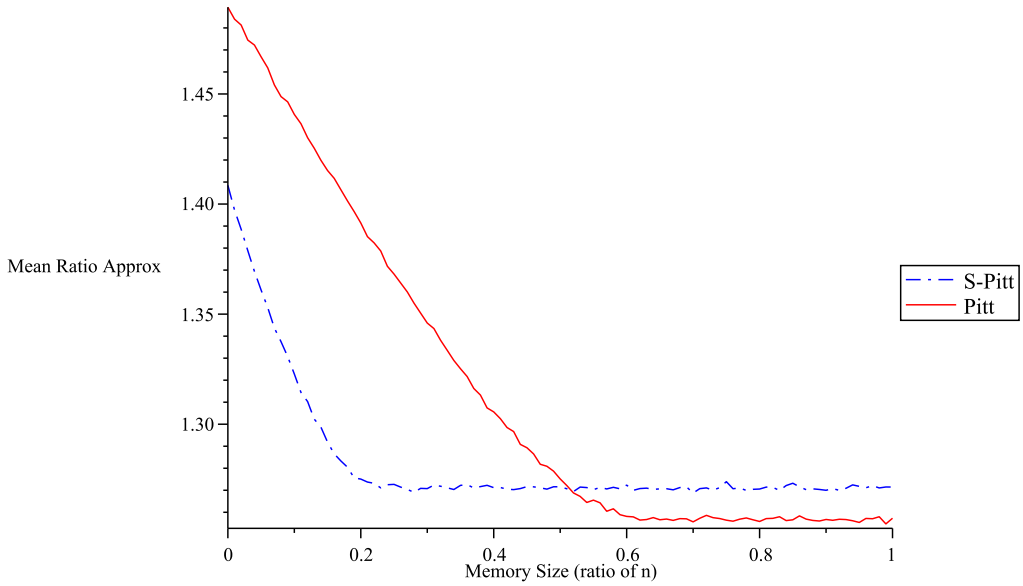


FIG. 3.11 – Courbes montrant les résultats des expérimentations effectuées pour les algorithmes Pitt et S-Pitt sur des chemins

3.5 Bilan

Nous avons étendu le modèle considéré précédemment en autorisant n bits d'espace mémoire. Cela nous a permis d'adapter et d'étudier d'autres algorithmes.

Nous avons notamment pu implémenter l'algorithme OT, en lui donnant la possibilité d'effectuer plusieurs passes. Nous avons montré que pour tout arbre, il existait un ordre de traitement des sommets tel qu'il n'en effectuait qu'une seule, et nous avons donné une borne maximale sur le nombre de passes qu'il pouvait effectuer. Nous avons aussi analysé sa complexité en nombre de requêtes, en montrant qu'il pouvait en effectuer moins de m mais qu'il pouvait aussi en effectuer de l'ordre de m^2.

Ensuite, nous avons adapté les algorithmes LR et ED et nous avons étudié leurs complexités. Nous avons montré que, s'il retournait la même couverture que les autres algorithmes, LR était celui qui effectuait le moins de requêtes dans ce modèle (où tous les algorithmes effectuent $d(u)$ requêtes pour les sommets u qui n'appartiennent pas à la solution) puisqu'il n'effectue pas de requête pour les sommets qu'il place dans la couverture. Nous avons aussi montré que LR et ED ne pouvaient pas réaliser plus de m requêtes.

Enfin, nous avons implémenté un algorithme probabiliste : Pitt. Nous avons tout d'abord analysé sa complexité en nombre de requêtes. Nous avons montré qu'il pouvait avoir le même comportement que LR (c'est-à-dire effectuer zéro requête pour les sommets qu'il place dans la solution), mais qu'il pouvait aussi effectuer (dans de rares cas) $2m$ requêtes (c'est-à-dire récupérer tous les voisins de chaque sommet, ce qui correspond au pire des cas possible). Nous avons ensuite analysé la qualité

des solutions qu'il construisait en fonction de la quantité de mémoire qu'il avait à sa disposition. Nous avons vu qu'il pouvait être très mauvais (c'est-à-dire retourner la quasi-totalité des sommets du graphe) sur certains graphes en ayant pourtant la possibilité de stocker en mémoire la moitié des sommets. Nous avons alors proposé un autre algorithme, S-Pitt, qui, profitant d'une hypothèse sur l'ordre de récupération des arêtes, gèrait mieux la mémoire. Nous avons vu cependant qu'il n'était pas toujours meilleur que l'algorithme Pitt. Le fait de ne pas pouvoir stocker tous les sommets du graphe peut considérablement dégrader les solutions produites (c'est en tout cas ce que nous avons observé avec l'algorithme Pitt).

Une première perspective de ce chapitre serait d'exprimer de façon plus précise le nombre de passes effectuées par l'algorithme OT, en le liant par exemple à un paramètre (une propriété) sur les arbres. Cela permettrait ensuite d'étudier de manière plus fine sa complexité en nombre de requêtes.

Une deuxième perspective serait d'exprimer de manière plus précise les complexités des différents algorithmes analysés dans ce chapitre, sans qu'elles ne soient en relation avec les solutions construites. Cela passerait notamment par une étude plus précise de la qualité des solutions produites par ces algorithmes, puisque ces deux critères sont liés.

Une autre perspective serait d'étendre le type d'étude menée sur l'algorithme Pitt aux autres algorithmes, en étudiant leurs performances dans un contexte de mémoire restreinte (par exemple, si l'algorithme OT disposait seulement d'une mémoire de taille $\frac{n}{2}$, de combien serait-il éloigné de l'optimal ?), afin de confirmer (ou de réfuter) le constat établi pour l'algorithme Pitt.

Chapitre 4

Etude expérimentale sur de gros graphes

Après avoir analysé de façon théorique les différents algorithmes présentés dans ce document, nous avons mené une étude expérimentale : ce chapitre y est consacré. Dans ce qui va suivre, nous allons décrire les caractéristiques des expérimentations réalisées puis nous présenterons et analyserons les résultats que nous avons obtenus.

4.1 Description générale

L'objectif des expérimentations que nous avons réalisées et qui sont décrites dans ce chapitre était de mettre à l'épreuve les différents algorithmes que nous avons étudiés sur de gros graphes. En effet, au vu des éléments présentés jusqu'à présent, une question naturelle qui se pose est de savoir combien de temps ils mettent concrètement pour calculer des couvertures sur de gros graphes.

Mais, dans un premier temps, qu'entend-on par « gros graphes » ? Nous éclaircirons ce point dans la section 4.1.4. Nous allons tout d'abord fournir les éléments techniques globaux de nos expériences ainsi que les algorithmes étudiés et les types de graphes utilisés.

4.1.1 Eléments techniques globaux

Nous avons choisi de nous baser en grande partie sur le modèle de traitement que nous avons considéré jusqu'à présent (voir la section 1.2), puisque les algorithmes que nous allons mettre à l'épreuve ont été conçu pour respecter les contraintes de ce modèle.

Cependant, en raison des coûts de mise en place d'un entrepôt de données (et aussi par souci de facilité de fabrication des graphes), nous avons préféré stocker les graphes que l'on a générés sur un disque dur externe.

Les graphes lus en entrée par la machine de traitement (un ordinateur standard) et les résultats générés sont stockés sur un gros disque dur externe : les algorithmes (qui s'exécutent sur la ma-

chine de traitement) lisent les graphes (stockés à distance) morceau par morceau et calculent leurs solutions au fur et à mesure, qu'ils écrivent « à la volée » sur le disque externe.

Nous donnons dans ce qui suit les caractéristiques techniques des différents éléments utilisés pour réaliser nos expériences.

La machine de traitement. Il s'agit d'un ordinateur portable doté d'un processeur double cœur cadencé à 2,8 GHz, d'une mémoire cache de 6 Mo et de 4 Go de mémoire vive.

Le disque dur externe. Il s'agit d'un disque dur USB 2.0[1] de 2 To, cadencé à 7 200 tours/minute et doté de 8 Mo de cache.

Caractéristiques logicielles. Le système d'exploitation installé sur la machine de traitement est une openSuSe 11.1 (Linux). Les différents programmes ont été codés en langage C, norme C99, afin d'utiliser le type de données `unsigned long long`, ainsi que les fonctions associées (pour la gestion des fichiers notamment).

A ce jour, plusieurs librairies ont été développées pour traiter de grandes instances, la plus répandue étant STXXL [50]. Cette librairie, élaborée dans le contexte des algorithmes I/O-efficient, permet d'effectuer de manière efficace des opérations sur de la mémoire externe (des disques durs par exemple). Elle est donc adaptée pour manipuler de grandes quantités de données qui ne tiennent pas dans la mémoire vive d'une machine.

Nous ne l'avons toutefois pas utilisée, en raison du fait que les algorithmes que nous avons implémentés se contentent de lire le graphe en entrée (puisque l'on ne peut pas le modifier) et d'écrire la solution en sortie (puisque l'on ne peut pas la consulter).
Ces deux contraintes s'avèrent en pratique réalistes, puisque la création d'un graphe de grande taille est coûteuse (si l'on veut exécuter plusieurs algorithmes sur un graphe, il faut donc préserver son intégrité) et puisque le fait de lire la solution durant l'exécution des algorithmes augmente de façon non négligeable leurs temps d'exécution.

Partant de ces contraintes fortes, nous avons choisi de lire le graphe comme un flux de données. C'est la raison pour laquelle nous avons privilégié l'utilisation de fichiers binaires avec les fonctions (assez bas niveau) fournies par le langage C.

Nous allons décrire de manière générale la façon dont les graphes sont stockés et la façon dont ils sont parcourus.

Le stockage des graphes

Il existe plusieurs façons de stocker un graphe : à l'aide d'une matrice d'adjacence, d'une liste d'adjacence, *etc.* Nous nous sommes basés sur la méthode décrite dans [7] (se reporter à la section 3.2 page 20). Nous avons donc opté pour une solution dans laquelle les graphes sont stockés à l'aide de deux fichiers.

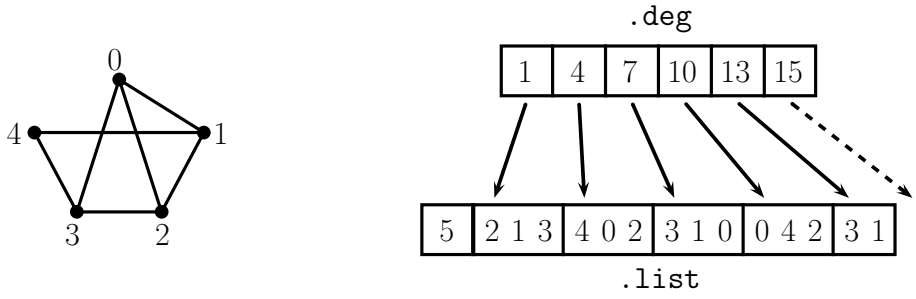
[1] Le taux de transfert d'un disque dur USB 2.0 peut atteindre au maximum 480 Mbits par seconde.

Le premier fichier contient $2m + 1$ valeurs, à savoir le nombre de sommets du graphe (indispensable pour créer un tableau de n bits lors de l'initialisation de certains algorithmes), puis la succession des voisins des différents sommets. Nous appelerons par la suite ce fichier `.list`.

Le second fichier contient $n + 1$ valeurs : elles permettent d'accéder aux voisins d'un sommet et de calculer son degré. Nous appelerons par le suite ce fichier `.deg`.

De manière plus précise, les sommets sont étiquetés de 0 jusqu'à $n - 1$.
Le fichier `.list` contient d'abord la valeur n, puis les sommets de $N(0)$, puis ceux de $N(1)$, *etc.* (toutefois, les voisins de chaque sommet sont stockés dans un ordre quelconque, pas forcément dans l'ordre des labels).
Le fichier `.deg` contient alors, dans l'ordre des labels, la position du premier voisin pour chaque sommet. Il contient $n+1$ valeurs, afin de permettre le calcul du degré du dernier sommet (la dernière valeur pointe donc sur la fin du fichier `.list`). En effet, pour calculer le degré du sommet i, on soustrait la $i^{\text{ème}}$ valeur de la $(i + 1)^{\text{ème}}$.

La figure 4.1 montre un exemple d'un tel stockage sur un petit graphe.



(a) Exemple de graphe (b) Stockage du graphe sur le disque

FIG. 4.1 – Stockage d'un graphe à 5 sommets et 7 arêtes à l'aide des fichiers `.deg` et `.list`

La lecture des graphes

Les algorithmes parcourent les graphes en lisant les deux fichiers décrits précédemment. Le fichier `.list` est d'abord lu pour connaître le nombre de sommets à traiter. Ensuite, le fichier `.deg` est lu pour connaître la position du premier voisin du premier sommet ainsi que la position du premier voisin du deuxième sommet (il y a donc deux pointeurs qui se suivent et qui permettent de délimiter l'ensemble des voisins d'un sommet). Une fois que ces positions sont connues, la lecture se poursuit dans le fichier `.list`, où sont récupérés les voisins du premier sommet. Si l'unité de traitement n'a pas besoin de tous les récupérer (par exemple, pour `LL`, si l'on trouve un voisin droit au bout du deuxième et qu'il en reste encore cinq), elle peut « enjamber » ceux qui restent et directement passer aux voisins du deuxième sommet. Elle procède de la même façon pour les sommets suivants, jusqu'à traiter les n sommets du graphe.

Si les algorithmes ont besoin de connaître les degrés des voisins, ils lisent de façon indépendante le fichier `.deg`. Ils disposent pour cela d'une autre tête de lecture qu'ils peuvent déplacer librement dans ce fichier, afin de récupérer les deux valeurs successives nécessaires au calcul (ils utilisent une autre variable de type `FILE*`).

Dans tous les cas, les algorithmes lisent le fichier `.list` de façon continue, comme un flux de données, à la différence qu'il peuvent « faire des bonds en avant » pour sauter les voisins des sommets qu'ils n'ont pas besoin de récupérer.
Les algorithmes qui n'ont pas besoin de connaître les degrés des voisins se contentent de lire le fichier `.deg` de façon continue. Ceux qui ont en besoin procèdent de la même manière pour parcourir le graphe, mais ils disposent d'une tête de lecture supplémentaire sur ce fichier, qu'ils peuvent déplacer comme ils veulent pour calculer ces degrés.

L'écriture des solutions

De manière simple, une couverture est un fichier qui contient la suite des sommets (représentés par leurs labels) qui la constituent. Elle est construite au fur et à mesure : dès qu'un algorithme décide qu'un sommet fait partie de la solution, il l'écrit dans le fichier. Chaque sommet n'apparaît qu'une fois puisque, comme nous l'avons vu dans les chapitres précédents, nos algorithmes ont été conçus pour ne pas produire de doublons.

4.1.2 Algorithmes implémentés

Nous avons testé six des algorithmes que nous avons présentés dans ce document, à savoir LR, ED, S-Pitt, LL, SLL et ASLL.

Par rapport à la manière dont les graphes sont fabriqués (voir la section 4.1.3), nous avons donné la possibilité aux algorithmes LL, SLL et ASLL d'appliquer une permutation circulaire aléatoire sur les labels (en appliquant une fonction $f(l) = l + \alpha \mod n$ sur chaque label l du graphe, avec $\alpha > 0$ une valeur aléatoire choisie au préalable). En effet, si nous avions mis en place l'aspect aléatoire des différents étiquetages possibles d'un graphe à sa création plutôt qu'à l'exécution de ces trois algorithmes, cela aurait entraîné des coûts de fabrication et de stockage supplémentaires.

Certes, le fait d'appliquer une permutation circulaire aléatoire ne permet pas de considérer de manière équiprobable les $n!$ étiquetages possibles d'un graphe, mais elle peut être réalisée de façon efficace durant les exécutions des algorithmes, sans utiliser de mémoire supplémentaire (mis à part le décalage α que l'on doit conserver).

Pour réaliser une véritable permutation sur les labels d'un graphe, il faudrait probablement mettre en œuvre des mécanismes plus complexes, qui nécessiteraient sans doute l'utilisation de mémoire supplémentaire.

Fidèles aux descriptions que nous en avons données dans le chapitre 3, les algorithmes LR et ED utilisent un tableau de n bits dans la mémoire de l'unité de traitement.

L'algorithme S-Pitt, qui profite naturellement du fait que les sommets (mais pas leurs voisins) soient stockés par ordre croissant des labels, utilise lui aussi un tableau de n bits, ce qui diffère quelque peu de la description fournie par l'implémentation 5 page 94. Nous avons en effet choisi de ne pas mettre en place une mémoire de taille variable car cela aurait nécessité, dans le pire des cas, plus que n bits mémoire.

Les algorithmes SLL et ASLL utilisent la seconde tête de lecture sur le fichier `.deg` pour récupérer les degrés des voisins (les quatre autres algorithmes n'en ont pas l'utilité). Sur un même graphe (de taille importante), nous avons remarqué que ces deux algorithmes mettaient plus de temps à s'exécuter que les autres. Nous avons alors mis au point des versions alternatives qui chargeaient au départ le fichier `.deg` en mémoire (afin de les distinguer des versions classiques, nous les appelerons SLL-M et ASLL-M). Nous avons constaté que leurs exécutions étaient plus rapides. En effet, le fait de devoir déplacer la tête de lecture pour calculer le degré de chaque voisin dans le fichier `.deg` augmente de manière non négligeable le nombre d'opérations sur le disque externe. Par conséquent, lorsque cela était possible, c'est-à-dire lorsque nous pouvions stocker en mémoire $64(n+1)$ bits[2], nous avons utilisé SLL-M et ASLL-M.

Dans le chapitre 2, nous avons défini la notion de requête comme étant le fait de récupérer un voisin d'un sommet (avec ou sans son degré) et nous nous sommes basés sur cette notion pour exprimer la complexité de nos algorithmes. Or, tel que nous les avons implémentés, il y a en réalité deux types de requêtes : avec ou sans récupération du degré. Nous n'allons toutefois pas redéfinir cette notion, mais nous réutiliserons ce constat dans la section 4.2, lorsque nous analyserons les résultats obtenus sur différents graphes.

4.1.3 Familles de graphes utilisées

Jusqu'à présent, nous avons supposé que les graphes étaient déjà disponibles sur la machine distante. Cependant, pour réaliser nos expérimentations, nous avons du les fabriquer.

La génération de graphes de grande taille est un problème à part entière, que nous n'aborderons pas dans les détails, mais nous donnerons tout de même des éléments afin de décrire les méthodes de génération utilisées et justifier les familles de graphes choisies.

Nous avons généré pour nos expériences des *ButterFly* [124], des graphes de *de Bruijn* [45, 71], des grilles, des hypercubes, des graphes bipartis complets et une extension de ces graphes que nous avons appelée *SpecialDense*. Nous avons choisi ces graphes pour plusieurs raisons.

- Ils peuvent facilement être fabriqués « à la volée ».
- Ils ont une description mathématique bien précise et peuvent donc être directement créés et utilisés par d'autres chercheurs, sans qu'il soit nécessaire de les mettre à disposition sur un site dédié.

[2]Les valeurs utilisées dans nos programmes (à savoir les labels et les degrés) sont de type `unsigned long long` : elles sont codées chacune sur 8 octets (c'est-à-dire 64 bits).

- Nous sommes capables, pour la plupart d'entre eux, d'exprimer de manière exacte la taille de leurs couvertures optimales.
- Il s'agit pour la plupart de graphes bipartis : la taille de leurs couvertures optimales ne dépassent pas la moitié des sommets, ce qui est intéressant lorsque l'on veut évaluer et comparer la qualité des solutions retournées.

Nous avons aussi utilisé un générateur de graphes aléatoires en loi de puissance [119], que nous avons rebaptisé `gengraph`.

Ces différentes familles de graphes se répartissent en deux catégories : les graphes denses, pour lesquels le nombre d'arêtes est élevé par rapport au nombre de sommets, et les graphes peu denses, pour lesquels le ratio $\frac{m}{n}$ est en $\mathcal{O}(1)$. Parmi les graphes peu denses, on retrouve les *ButterFly*, les graphes de *de Bruijn*, les grilles et (bien souvent) les graphes aléatoires en loi de puissance. Et parmi les graphes denses, on retrouve les hypercubes, les graphes bipartis complets et les *SpecialDense*.

Nous allons donner dans ce qui va suivre les caractérisques globales de ces différents types de graphe, ainsi que la manière dont nous les avons générés.

De façon générale, nous avons fabriqué les graphes de la manière suivante : pour chaque sommet, nous avons calculé localement leurs voisins et nous les avons écrits au fur et à mesure. Pour cela, nous avons directement utilisé leurs labels. Cette méthode simpliste, qui implique qu'un graphe d'une taille donnée est toujours étiqueté de la même façon, nous a permis de générer des graphes de l'ordre de plusieurs centaines de Go sur disque. Toutefois, lorsque cela était possible, nous appliquions une permutation aléatoire sur les voisins de chaque sommet, afin qu'ils ne soient pas toujours stockés dans le même ordre dans le fichier `.list`.

Les *ButterFly*

Par rapport à la description fournie dans [124], nous avons considéré une version non orientée de ces graphes et nous nous sommes restreints à la base 2, qui est très simple à implémenter en langage C avec les opérateurs binaires.

Soit $BF_d = (V, E)$ un *ButterFly* non orienté de dimension d (et de base 2). Ce graphe possède $n = (d+1) \cdot 2^d$ sommets et $m = d \cdot 2^{d+1}$ arêtes ; sa densité est faible, puisque $\frac{m}{n} < 2$.

Dans un *ButterFly* de dimension d, chaque sommet est représenté par une paire (w, l) où w est un nombre binaire à d bits et l un entier compris entre 0 et d (dans nos programmes, les labels sont obtenus en additionnant w et $l \cdot 2^d$). Deux sommets (w, l) et (w', l') sont reliés par une arête si et seulement si $l' = l + 1$ et si w' est identique à w, à l'exception éventuelle du $(l+1)^{\text{ème}}$ bit en partant de la gauche. Par exemple, le sommet $(01, 1)$ a pour voisin $(01, 2)$, $(00, 2)$, $(01, 0)$ et $(11, 0)$.

De ce fait, les *ButterFly* sont des graphes bipartis particuliers constitués de $d+1$ couches de 2^d sommets chacune, dans lesquelles chaque sommet possède 2 voisins dans la couche précédente (sauf pour la première) et 2 voisins dans la couche suivante (sauf pour la dernière). Par conséquent, tous les sommets des couches intermédiaires (il y en a exactement $(d-1) \cdot 2^d$) sont de degré 4, tandis que ceux des deux couches extérieures (il y en a exactement 2^{d+1}) sont de degré 2.

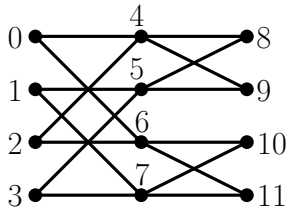


FIG. 4.2 – *ButterFly* de dimension 2

La figure 4.2 montre un exemple de *ButterFly*.

La taille d'une couverture optimale dans un *ButterFly* de dimension d est égale à $\lfloor \frac{d+1}{2} \rfloor \cdot 2^d$. Il s'agit de la taille du plus petit sous-ensemble de sommets dans ce graphe biparti. Par exemple, sur le graphe de la figure 4.2, la couverture optimale est constituée des sommets 4, 5, 6 et 7.

Les graphes de *de Bruijn*

Cette famille de graphe est utilisée dans plusieurs branches de l'informatique et dans de nombreuses applications en général (voir par exemple [131]).

Tout comme pour les *ButterFly*, nous avons considéré une version simple et non orientée des graphes de *de Bruijn* et nous nous sommes restreints à la base 2, là aussi pour des raisons de facilité de programmation.

Soit $dB_d = (V, E)$ un graphe de *de Bruijn* simple et non orienté de dimension d (et de base 2). Ce graphe possède $n = 2^d$ sommets et $m = 2^{d+1} - 3$ arêtes ; sa densité est faible, puisque $\frac{m}{n} < 2$.

Dans un graphe de *de Bruijn*, un sommet u est relié par une arête à un sommet v si et seulement si la représentation binaire de son étiquette est décalée d'un bit vers la gauche (le bit de poids faible est alors égal à 0 ou à 1). Par exemple, le sommet 010 a pour voisin 100, 101 et 001.
Pour $d > 2$, il est constitué de $n - 4$ sommets de degré 4, de deux sommets de degré 3 et de deux sommets de degré 2 (il est donc presque régulier).

La figure 4.3 montre un exemple de graphe de *de Bruijn*.

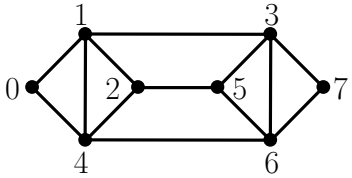


FIG. 4.3 – Graphe de *de Bruijn* simple et non orienté, de base 2 et de dimension 3

Plusieurs travaux ont été menés récemment pour tenter d'évaluer la taille maximale d'un ensemble indépendant sur les graphes de *de Bruijn* (voir par exemple [34]). Il a été établi dans [88] que cette valeur tendait vers $\frac{n}{2}$. Nous en déduisons donc que la taille d'une couverture minimale dans un dB_d est d'au moins 2^{d-1}.

Les grilles

Ces graphes, que nous avons déjà utilisés dans le chapitre 2, peuvent eux aussi être générés de manière efficace.

Soit $GR_{p\times q} = (V, E)$ une grille constituée de p lignes et de q colonnes de sommets. Ce graphe possède $n = p \times q$ sommets et $m = 2pq - p - q$ arêtes ; sa densité est faible, puisque $\frac{m}{n} < 2$.

De façon générale, chaque sommet est connecté au sommet de la colonne précédente, à celui de la colonne suivante, au sommet de la ligne précédente et à celui de la ligne suivante. Il y a donc une majorité de sommets de degré 4 (il y en a exactement $(p-2)(q-2)$), $2(p+q-4)$ sommets de degré 3 et quatre sommets de degré 2.

Les grilles sont des graphes bipartis. Comme pour les *ButterFly*, la taille d'une couverture optimale correspond à la taille du plus petit sous-ensemble de sommets. La taille d'une couverture optimale dans un $GR_{p\times q}$ est donc de $\left\lfloor \frac{pq}{2} \right\rfloor$.

Les hypercubes

Soit $H_d = (V, E)$ un hypercube de dimension d. Ce graphe possède $n = 2^d$ sommets et $m = d \cdot 2^{d-1}$ arêtes ; sa densité est de $\frac{d}{2}$.

Dans un hypercube, chaque sommet possède d voisins. En effet, deux sommets sont adjacents si les représentations binaires de leurs étiquettes (codées sur d bits) ne diffèrent que d'un seul bit.

La figure 4.4 montre un exemple d'hypercube.

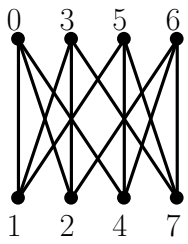


FIG. 4.4 – Hypercube de dimension 3

D'après le théorème de Hall [74], les hypercubes sont des graphes bipartis qui admettent des couplages parfaits. La taille d'une couverture optimale correspond donc à la moitié des sommets présents dans ce couplage (c'est-à-dire à la moitié des sommets du graphe). Donc, dans un hypercube de dimension d, la taille d'une couverture optimale est égale à 2^{d-1}.

Les graphes bipartis complets

Tout comme les grilles, nous avons déjà évoqués ces graphes dans le chapitre 2.

Soit $K_{a,b} = (X \cup Y, E)$ un graphe biparti complet avec $a < b$. Ce graphe possède $n = a + b$ sommets ($a = |X|$ et $b = |Y|$) et $m = ab$ arêtes ; sa densité est de $\frac{ab}{a+b}$.

Dans ce graphe, les a sommets de X sont de degré b et les b sommets de Y sont de degré a.

Il est facile de voir que les a sommets de l'ensemble X dans un $K_{a,b}$ constituent une couverture de taille minimum.

Les *SpecialDense*

Il s'agit d'une extension des graphes bipartis complets, dans lesquels on transforme l'ensemble des sommets X en clique[3].

Soit $K^*_{a,b} = (X \cup Y, E)$ un *SpecialDense* avec $a < b$. Ce graphe possède $n = a + b$ sommets ($a = |X|$ et $b = |Y|$) et $m = \frac{a(a-1)}{2} + ab$ arêtes (sa densité est de $\frac{a(a-1+2b)}{2(a+b)}$). Dans ce graphe, les a sommets de X sont de degré $a - 1 + b = n - 1$ et les b sommets de Y sont de degré a. La figure 4.5 montre un exemple de *SpecialDense*.

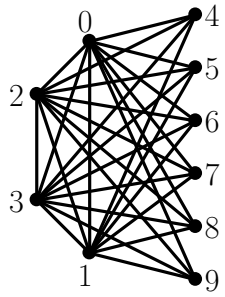


FIG. 4.5 – *SpecialDense* à $4 + 6$ sommets

Comme pour les graphes bipartis complets, la couverture de taille minimum est constituée des a sommets de X.

Nous avons choisi ces graphes parce que ce sont ceux qui avaient la plus forte densité et pour lesquels nous pouvions donner de manière exacte la taille d'une couverture optimale.

Les graphes aléatoires en loi de puissance

Nous avons aussi exécuté nos algorithmes sur des graphes aléatoires en loi de puissance, c'est-à-dire des graphes pour lesquels la répartition des degrés des sommets suit une loi de puissance. Pour cela, nous avons repris le générateur de *F. Vigier et M. Latapy* [119], que nous avons rebaptisé `gengraph`. Ce générateur se base sur le modèle de *M. Molloy et B. Reed* [93]. Il s'agit probablement d'un des générateurs les plus performants pour produire ce type de graphes.

Ce générateur est composé de deux programmes.
Le premier programme prend en entrée un nombre de sommets, un degré minimum, éventuellement un degré maximum (dans le cas contraire, il n'est pas borné) et un exposant. En fonction de

[3]Dans une *clique*, chaque sommet est relié par une arête à chacun des autres sommets.

ces paramètres, il génère une distribution des degrés pour un graphe simple et connexe (si on le souhaite).
Le second programme génère un graphe à partir d'une distribution des degrés créée à l'aide du premier programme.

Nous nous sommes contentés de modifier la manière dont étaient écrits les graphes dans le second programme, afin qu'ils soient de même format que ceux que l'on générait.

Pour générer nos graphes, nous avons spécifié des degrés minimaux compris entre 1 et 3 et nous avons utilisé des exposants compris entre 2,1 et 2,5. Nous n'avons pas borné les degrés maximaux et nous n'avons pas activé le test de connexité.

Ce générateur ne crée pas les graphes « à la volée » (il a besoin d'un espace mémoire linéaire en la taille du graphe), mais il permet tout de même de fabriquer assez rapidement des graphes de l'ordre de plusieurs dizaines de millions de sommets et d'arêtes[4].

4.1.4 Démarche expérimentale

Nous allons à présent décrire la démarche que nous avons employée pour mener à bien nos expérimentations sur de gros graphes.

Nous avons vu au début du chapitre 2 que la notion de *memory-efficient* était relative puisque son interprétation différait d'une application à une autre. Il en est de même avec la notion de gros graphes. En effet, suivant que l'on étudie des algorithmes exponentiels, que l'on fasse de la programmation linéaire ou bien que l'on manipule des algorithmes de complexité linéaire, les limites sur la taille des graphes ne sont pas situées au même endroit. C'est la raison pour laquelle nous avons défini plusieurs niveaux relatifs à la taille des graphes.

Pour quantifier la taille des graphes, nous avons considéré à la fois leurs nombres de sommets et leurs nombres d'arêtes, puisqu'ils n'ont pas tous la même densité et puisque ces deux critères sont conjointement liés à l'espace qu'ils prennent sur le disque.

Nous décrivons à présent les différents niveaux de taille que nous avons considérés.

1$^{\text{er}}$ niveau. La taille des graphes $(n + m)$ atteint 400 000 (concrètement, de la manière dont nous les stockons, ces graphes prennent quelques Mo sur le disque). Dans ce niveau, sur les graphes peu denses (qui contiennent plus de 50 000 sommets), les solveurs de programmation linéaire relaxée (`LPSolve` par exemple) peuvent atteindre leurs limites en terme de mémoire utilisée et de temps d'exécution sur les machines actuelles.

2$^{\text{ème}}$ niveau. La taille des graphes atteint 200 millions (concrètement, ils prennent chacun quelques Go sur le disque). Il s'agit de la taille maximale atteignable sur notre machine pour construire des graphes en loi de puissance avec `gengraph`.

[4]Pour le plus gros, nous avons pu créer en moins de cinq heures sur notre machine un graphe à 30 millions de sommets et 96 millions d'arêtes.

3ème niveau. La taille des graphes atteint 30 milliards (concrètement, ils prennent chacun quelques centaines de Go sur le disque). Par rapport aux deux niveaux précédents, on commence à atteindre des tailles que l'on peut qualifier (de notre point de vue) de conséquentes. De plus, sur des graphes peu denses (qui ont plus de 8 milliards de sommets), il n'est plus possible d'utiliser les algorithmes SLL-M et ASLL-M.

4ème niveau. La taille des graphes atteint 100 milliards (concrètement, ils prennent environ 1,5 To sur le disque). Avec notre machine (qui dispose de 4 Go de mémoire vive), nous ne pouvons plus exécuter les algorithmes qui requièrent n bits mémoire sur des graphes peu denses (qui contiennent plus de 30 milliards de sommets). Ce niveau correspond donc aux limites atteignables avec notre matériel.

Critères évalués. Nous nous sommes intéressés à la qualité des solutions produites par nos algorithmes, ainsi qu'au nombre de requêtes effectuées. Nous avons aussi mesuré à partir du troisième niveau leurs temps d'exécution. Pour cela, nous avons utilisé la commande `/usr/bin/time` de UNIX, qui fournit les valeurs suivantes (elles sont exprimées en secondes) :

- le temps réel écoulé, entre le moment où on lance l'exécution du programme et le moment où il se termine ;
- le *user time*, qui correspond au temps dépensé par le processeur pour réaliser les instructions du programme ;
- le *system time*, qui correspond au temps dépensé par le processeur pour réaliser les opérations systèmes (typiquement les entrées/sorties) liées au programme.

Pour des raisons de précision et d'équité, nous nous baserons uniquement sur le *user time* et sur le *system time*. En effet, comme le système d'exploitation de notre machine de traitement (standard) exécute toujours d'autres processus en tâche de fond, on peut avoir deux exécutions d'un même programme avec les mêmes entrées qui présentent des temps d'exécution (réels écoulés) différents.

Déroulement des expérimentations

De manière générale, nous avons exécuté une ou plusieurs fois les algorithmes sur chaque graphe. A la fin des exécutions, nous avons conservé, pour chaque algorithme, la meilleure des solutions en terme de qualité. Nous avons alors conservé le nombre de requêtes effectuées pour construire cette solution.

Par rapport à la manière dont les graphes sont construits, les algorithmes LR et ED sont déterministes. Il n'est donc pas utile de les exécuter plusieurs fois. En revanche, pour l'algorithme S-Pitt, qui est par nature probabiliste, ainsi que pour les algorithmes LL, SLL et ASLL, qui peuvent générer une permutation circulaire aléatoire sur les labels, on peut espérer améliorer la qualité des solutions produites en les exécutant de nouveau.

Pour les deux premiers niveaux, pour lesquels les temps d'exécution des algorithmes sont rapides, nous avons exécuté les algorithmes LR et ED une fois et les algorithmes S-Pitt, LL, SLL et ASLL 10

(resp. 5) fois au premier (resp. deuxième) niveau.

A partir du troisième niveau, nous nous sommes mis en quelque sorte dans la peau d'un utilisateur qui disposerait de ressources (temps et espace disque) limitées. Ainsi, sur chaque graphe, nous avons exécuté les six algorithmes une première fois, sans appliquer de permutation circulaire aléatoire sur les labels pour LL, SLL et ASLL. En fonction des résultats obtenus en terme de qualité de solution et en fonction du temps et de l'espace disque déjà dépensés, nous avons exécuté de nouveau certains algorithmes un certain nombre de fois (la plupart du temps une fois), en appliquant une permutation circulaire aléatoire sur les labels le cas échéant.

Pour le quatrième niveau, nous avons voulu atteindre les limites techniques de notre matériel. Pour cela, nous avons créé deux graphes : un graphe peu dense (un *ButterFly* de dimension 30) sur lequel les algorithmes LR, ED et S-Pitt ne peuvent pas s'exécuter sur notre machine, et un graphe dense (un biparti complet à $250\ 000 + 380\ 000$ sommets) sur lequel nous avons pu exécuter LR, ED, S-Pitt, LL, SLL-M et ASLL-M.

4.2 Résultats et interprétation

Nous allons tout d'abord donner dans cette section les résultats que nous avons obtenus lors de nos expériences. Ensuite, nous les interpréterons.

4.2.1 Présentation des résultats

Nous allons présenter les résultats par niveau de taille. Pour les niveaux 1 et 2, comme les temps d'exécution des algorithmes sont rapides, nous donnerons un tableau par critère étudié, à savoir la qualité des solutions et la complexité en nombre de requêtes. Ces tableaux regrouperont l'ensemble des résultats obtenus pour chaque algorithme sur chaque instance étudiée. Nous donnerons plus de détails pour le troisième et pour le quatrième niveau.

Résultats obtenus au premier niveau

Nous avons généré une instance pour chaque type de graphe décrit dans la section précédente. Nous avons aussi généré sept graphes aléatoires avec `gengraph`. Le tableau 4.1 présente les caractéristiques des six instances que nous avons générées nous-même. Le tableau 4.2 présente les caractéristiques des sept graphes aléatoires en loi de puissance générés avec `gengraph`.

Ces différents graphes prennent entre 1,44 et 4,13 Mo sur le disque dur.

Le tableau 4.3 montre les résultats obtenus en terme de qualité de solution sur les différentes instances présentées dans les tableaux 4.1 et 4.2. Les nombres en gras correspondent, pour chaque instance, à la meilleure valeur obtenue sur les six algorithmes. A l'inverse, les nombres en italiques signifient que les valeurs sont très mauvaises (lorsque l'algorithme retourne quasiment tous les sommets du graphe).

Le tableau 4.4 montre les résultats obtenus en terme de complexité en nombre de requêtes sur les différentes instances présentées dans les tableaux 4.1 et 4.2. Les nombres en italiques signifient que les valeurs sont mauvaises (lorsque l'algorithme effectue m requêtes et plus).

L'exécution d'un des six algorithmes sur l'un de ces graphes prend en moyenne une dizaine de millisecondes. De plus, à ce niveau, il n'y a pas de réelles différences entre les temps d'exécution de SLL (resp. ASLL) et de SLL-M (resp. ASLL-M).

A titre comparatif, nous avons utilisé l'algorithme 2-approché classique basé sur une formulation du VERTEX COVER comme un programme linéaire. Rappelons qu'il est possible de formuler le VERTEX COVER à l'aide du programme linéaire en nombres entiers suivant :

– il y a une variable $X_u \in \{0, 1\}$ pour chaque sommet $u \in V$ du graphe ;
– il y a une contrainte $X_u + X_v \geq 1$ pour chaque arête $uv \in E$ du graphe.
→ Il s'agit de minimiser $\sum_{u \in V} X_u$.

L'algorithme approché consiste alors à résoudre la relaxation linéaire de ce programme (les variables X_u sont maintenant des variables réelles comprises entre 0 et 1), puis à mettre dans la couverture les sommets u tels que $X_u \geq \frac{1}{2}$.

Le tableau 4.5 présente les temps d'exécution et la qualité des solutions obtenues en utilisant `LPSolve` pour résoudre le programme linéaire sur trois des instances que nous avons considérées. On peut constater que, sur une même instance, nos algorithmes sont au minimum 140 fois plus rapides.

Instance	n	m	$\frac{n}{m}$ (densité)	OPT
`butterfly-13`	114 688	212 992	1,86	57 344
`debruijn-16`	65 536	131 069	1,99	$> 32\,768$
`grid-250.300`	75 000	149 450	1,99	37 500
`hypercube-14`	16 384	114 688	7	8 192
`compbip-300.450`	750	135 000	180	300
`spedens-275.400`	675	147 675	218,78	275

TAB. 4.1 – Caractéristiques des graphes générés pour le premier niveau

Instance[a]	n	m	$\frac{n}{m}$ (densité)	Δ
`rg-20000_3`	20 000	113 907	5,69	13 996
`rg-25000_2`	25 000	100 775	4,03	3 959
`rg-30000_1`	30 000	79 495	2,65	21 269
`rg-35000_3`	35 000	129 845	3,71	2 259
`rg-40000_2`	40 000	107 105	2,68	2 418
`rg-45000_1`	45 000	77 114	1,71	7 843
`rg-50000_1`	50 000	82 254	1,7	9 817

TAB. 4.2 – Caractéristiques des graphes aléatoires utilisés pour le premier niveau

[a]Le dernier chiffre mentionné dans le nom de l'instance indique le degré minimum du graphe.

Instance	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-13	**57 344**	*114 688*	87 885	98 308	90 112	98 309
debruijn-16	**36 411**	58 256	50 726	43 692	44 178	45 728
grid-250.300	**37 500**	*74 982*	60 856	*74 997*	*74 993*	*74 997*
hypercube-14	**8 192**	*16 330*	15 756	*16 371*	*16 374*	*16 375*
compbip-300.450	450	600	550	**300**	**300**	450
spedens-275.400	*674*	442	424	**275**	**275**	*674*
rg-20000_3	**7 415**	13 406	10 037	8 785	8 047	*19 999*
rg-25000_2	**8 792**	16 160	11 657	10 486	9 496	*24 967*
rg-30000_1	**2 171**	4 290	2 994	3 795	2 287	*29 999*
rg-35000_3	**18 559**	30 116	24 288	21 897	20 604	*34 953*
rg-40000_2	**17 493**	30 846	22 869	20 778	19 072	*39 600*
rg-45000_1	**8 929**	17 752	12 179	10 603	9 321	*44 870*
rg-50000_1	**9 668**	19 224	13 172	11 282	10 105	*49 877*

TAB. 4.3 – Résultats obtenus en terme de qualité de solution (1$^{\text{er}}$ niveau)

Instance	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-13	*212 992*	**129 166**	197 852	*229 784*	*229 786*	188 157
debruijn-16	116 495	**80 159**	114 202	*149 055*	*148 916*	*139 775*
grid-250.300	*149 450*	**73 725**	128 912	137 007	137 119	113 017
hypercube-14	*114 688*	**21 715**	42 745	33 450	33 954	33 905
compbip-300.450	*135 000*	**45 495**	60 771	*135 300*	*135 300*	*135 450*
spedens-275.400	**674**	64 408	69 672	110 350	110 350	5 321
rg-20000_3	51 496	**45 061**	59 584	59 759	70 120	50 854
rg-25000_2	44 526	40 719	53 915	54 166	63 103	**36 668**
rg-30000_1	48 691	47 534	52 839	**41 592**	55 749	59 324
rg-35000_3	59 196	**49 795**	73 589	82 288	90 569	57 844
rg-40000_2	54 710	**49 176**	68 906	71 928	83 224	59 764
rg-45000_1	44 981	**44 387**	53 545	59 464	60 131	65 771
rg-50000_1	50 481	**49 682**	60 226	62 566	63 849	77 748

TAB. 4.4 – Résultats obtenus en terme de complexité en nombre de requêtes (1$^{\text{er}}$ niveau)

Instance	Nombre de sommets	Rang	Temps d'exécution (*user + system time*)
debruijn-16	38 271	2$^{\text{ème}}$	2 114,14 sec $\approx$ 35 min
compbip-300.450	300	1$^{\text{er}}$	14,04 sec
rg-25000_2	10 854	4$^{\text{ème}}$	612,51 sec $\approx$ 10 min

TAB. 4.5 – Qualité des solutions et temps d'exécution obtenus avec `LPSolve` sur trois instances

Résultats obtenus au deuxième niveau

Comme pour le niveau précédent, nous avons généré une instance pour chaque type de graphe décrit dans la section précédente. Nous avons aussi généré sept graphes aléatoires avec `gengraph`. Le tableau 4.6 présente les caractéristiques des six instances que nous avons générées nous-même. Le tableau 4.7 présente les caractéristiques des sept graphes aléatoires en loi de puissance générés avec `gengraph`.

Ces différents graphes prennent entre 1,04 et 2,01 Go sur le disque dur.

Le tableau 4.8 montre les résultats obtenus en terme de qualité de solution sur les différentes instances présentées dans les tableaux 4.6 et 4.7. Les nombres en italiques signifient que les valeurs sont mauvaises (lorsque l'algorithme retourne quasiment tous les sommets ou lorsqu'il retourne bien plus de sommets que les autres).

Le tableau 4.9 montre les résultats obtenus en terme de complexité en nombre de requêtes sur les différentes instances présentées dans les tableaux 4.6 et 4.7. Les nombres en italiques signifient à l'inverse que les valeurs sont mauvaises (lorsque l'algorithme effectue m requêtes et plus).

L'exécution d'un des six algorithmes sur l'un de ses graphes prend en moyenne entre une vingtaine de secondes (sur `spedens-7500.12000`) et un peu plus d'une minute (sur `grid-6000.9000`). Les temps d'exécution des algorithmes SLL et ASLL peuvent prendre chacun entre une cinquantaine de secondes (sur `hypercube-23`) et plusieurs minutes (sur les graphes aléatoires en loi de puissance).

Instance	n	m	$\frac{n}{m}$ (densité)	OPT
`butterfly-21`	46 137 344	88 080 384	1,91	23 068 672
`debruijn-25`	33 554 432	67 108 861	1,99	$> 16\,777\,216$
`grid-6000.9000`	54 000 000	107 985 000	1,99	27 000 000
`hypercube-23`	8 388 608	96 468 992	11,5	4 194 304
`compbip-7000.15000`	22 000	105 000 000	4 772,73	7 000
`spedens-7500.12000`	19 500	118 121 250	6 057,5	7 500

TAB. 4.6 – Caractéristiques des graphes générés pour le deuxième niveau

Instance[a]	n	m	$\frac{n}{m}$ (densité)	Δ
`rg-20m_1`	20 000 000	59 624 494	2,98	9 684 355
`rg-20m_2`	20 000 000	90 808 193	4,54	5 823 864
`rg-25m_1`	25 000 000	70 911 180	2,84	3 218 406
`rg-25m_2`	25 000 000	87 837 432	3,51	2 441 204
`rg-30m_1`	30 000 000	82 356 722	2,75	4 553 669
`rg-30m_2`	30 000 000	81 819 916	2,73	1 006 976
`rg-35m_2`	35 000 000	96 555 269	2,76	1 853 607

TAB. 4.7 – Caractéristiques des graphes aléatoires utilisés pour le deuxième niveau

[a]Le dernier chiffre mentionné dans le nom de l'instance indique le degré minimum du graphe.

Instance	LR	ED	S-Pitt	LL	SLL-M	ASLL-M
butterfly-21	**23 068 672**	*46 137 344*	36 060 828	41 943 040	39 245 928	41 943 149
debruijn-25	**22 369 621**	29 826 162	26 022 017	22 406 058	24 395 861	27 643 823
grid-6000.9000	**27 000 000**	*53 999 424*	43 962 047	*53 999 997*	*53 999 993*	*53 999 997*
hypercube-23	**4 194 304**	*8 386 016*	8 326 123	*8 388 597*	*8 388 593*	*8 388 594*
compbip-7000.15000	15 000	14 000	13 856	**7 000**	**7 000**	15 000
spedens-7500.12000	*19 499*	12 026	11 971	**7 500**	**7 500**	*19 499*
rg-20m_1	**1 988 097**	3 930 592	2 681 231	9 859 431	2 084 910	*19 998 786*
rg-20m_2	**6 823 445**	12 578 120	9 120 903	9 859 421	7 377 191	*19 976 159*
rg-25m_1	**3 546 433**	7 022 208	4 726 633	10 249 991	3 723 041	*24 988 555*
rg-25m_2	**9 690 407**	17 496 468	12 821 807	12 031 339	10 529 231	*24 913 016*
rg-30m_1	**13 044 528**	22 984 450	17 151 200	17 706 253	14 230 885	*29 720 676*
rg-30m_2	**4 702 934**	9 314 526	6 329 866	9 294 126	4 939 477	*29 979 597*
rg-35m_2	**15 092 027**	26 654 266	19 876 649	18 795 856	16 442 224	*34 692 308*

TAB. 4.8 – Résultats obtenus en terme de qualité de solution ($2^{\text{ème}}$ niveau)

Instance	LR	ED	S-Pitt	LL	SLL-M	ASLL-M
butterfly-21	*88 080 384*	**53 301 140**	80 978 357	*90 988 251*	*92 270 593*	70 606 263
debruijn-25	44 739 241	**41 012 010**	58 404 532	*76 417 058*	*71 701 858*	*67 429 715*
grid-6000.9000	*107 985 000*	**53 917 286**	93 128 766	98 976 039	98 978 913	81 025 133
hypercube-23	*96 468 992*	**11 411 972**	22 738 282	16 916 570	16 963 774	16 892 962
compbip-7000.15000	*105 000 000*	**56 009 423**	57 026 526	*105 007 000*	*105 007 000*	*105 015 000*
spedens-7500.12000	**19 499**	56 064 236	56 486 081	90 009 469	90 009 469	204 842
rg-20m_1	29 288 946	28 613 225	32 914 621	**20 446 719**	35 306 991	34 420 719
rg-20m_2	36 535 474	33 530 551	44 253 789	34 689 835	51 367 285	**31 117 971**
rg-25m_1	32 218 927	31 607 734	37 671 723	**25 690 610**	41 264 920	33 385 504
rg-25m_2	39 526 683	**35 950 744**	48 707 048	45 504 990	57 534 137	36 753 205
rg-30m_1	41 252 284	**37 094 200**	51 699 001	47 673 779	62 403 566	43 065 067
rg-30m_2	36 701 126	36 015 480	43 233 530	**32 530 588**	47 680 910	37 010 761
rg-35m_2	48 414 005	**43 553 832**	60 539 903	60 366 907	73 012 142	51 475 230

TAB. 4.9 – Résultats obtenus en terme de complexité en nombre de requêtes ($2^{\text{ème}}$ niveau)

Résultats obtenus au troisième niveau

Nous avons généré une instance pour chaque type de graphe décrit dans la section précédente, puis nous avons exécuté chaque algorithme une fois, sans appliquer de permutation circulaire aléatoire (pour LL, SLL et ASLL). En fonction des résultats obtenus, des ressources (temps et espace disque) déjà dépensées, nous avons ensuite exécuté de nouveau certains algorithmes, en appliquant une permutation circulaire aléatoire sur les labels du graphe le cas échéant.

Le tableau 4.10 donne les caractéristiques des six instances que nous avons générées et sur lesquelles nous avons exécuté nos algorithmes.

Ces différentes instances prennent chacune plusieurs centaines de Go sur le disque dur (entre 225 et 320). Pour les graphes peu denses (butterfly-28, debruijn-33 et grid-75000.90000), le

fichier `.deg` prend en moyenne une cinquantaire de Go. Il prend 8 Go pour `hypercube-30`; il ne prend que quelques Mo pour `compbip-35000.500000` et `spedens-70000.180000`. Pour chacun de ces six graphes, le fichier `.list` prend environ 230 Go sur le disque.

Les tableaux 4.11, 4.12, 4.13, 4.14, 4.15 et 4.16 fournissent les résultats obtenus sur les six instances décrites dans le tableau 4.10.

Dans ces six tableaux, pour les algorithmes exécutés plusieurs fois (au delà d'une exécution, le nombre est indiqué entre parenthèses), nous avons conservé la taille de la meilleure solution, le nombre de requêtes effectuées pour obtenir cette solution ainsi que les moyennes des durées des différentes exécutions.

A titre indicatif, sur `compbip-35000.500000`, l'algorithme SLL-M, que l'on peut exécuter puisque la taille du fichier `.deg` (qu'il charge en mémoire) n'est que de quelques Mo, met 1 257,19 + 88,80 ≈ 23 minutes, ce qui est bien plus rapide que l'algorithme SLL (qui met 6H12).

A titre indicatif, sur `spedens-70000.180000`, l'algorithme SLL-M, que l'on peut exécuter puisque la taille du fichier `.deg` (qu'il charge en mémoire) n'est que de quelques Mo, met 907,29+68,34 ≈ 17 minutes, ce qui est bien plus rapide que l'algorithme SLL (qui met 4H28).

Instance	n	m	$\frac{n}{m}$ (densité)	OPT	Temps de création
`butterfly-28`	7 784 628 224	15 032 385 540	1,931	3 758 096 384	≈ 1H12
`debruijn-33`	8 589 934 592	17 179 869 183	2	> 4 294 967 296	≈ 1H23
`grid-75000.90000`	6 750 000 000	13 499 835 000	2	3 375 000 000	≈ 48 min
`hypercube-30`	1 073 741 824	16 106 127 360	15	536 870 912	≈ 59 min
`compbip-35000.500000`	535 000	17 500 000 000	32 710,28	35 000	≈ 1H07
`spedens-70000.180000`	250 000	15 049 965 000	60 199,86	70 000	≈ 1H04

TAB. 4.10 – Caractéristiques des graphes générés pour le troisième niveau

	LR	ED	S-Pitt (2)	LL (2)	SLL	ASLL
Qualité	**3 758 096 384**	7 516 192 768	6 130 985 368	7 258 269 582	7 247 757 312	7 516 192 768
Complexité	*15 032 385 536*	**9 305 823 421**	13 768 876 679	*15 558 102 744*	*15 300 847 900*	10 916 377 615
User time	2 929,04	3 026,20	3 368,41	3 431,24	6 966,95	5 478,17
System time	1 385,72	1 526,94	1 352,63	1 403,74	11 679,61	7 644,58
Total	≈ 1H12	≈ 1H16	≈ 1H19	≈ 1H21	≈ 5H11	≈ 3H39

TAB. 4.11 – Récapitulatif des résultats obtenus sur `butterfly-28`

	LR	ED	S-Pitt (2)	LL (2)	SLL	ASLL
Qualité	**5 726 623 061**	7 635 497 414	6 661 974 224	8 293 641 999	*8 589 934 589*	*8 589 934 591*
Complexité	11 453 246 121	**10 498 772 566**	14 951 011 468	*18 710 312 945*	*18 611 565 247*	15 748 248 250
User time	3 176,53	3 348,72	3 651,8	3 765,5	10 191,02	7 299,91
System time	1 660,22	1 701,56	1 522,36	1 609,26	17 641,58	11 217,32
Total	≈ 1H21	≈ 1H25	≈ 1H27	≈ 1H30	≈ 7H44	≈ 5H09

TAB. 4.12 – Récapitulatif des résultats obtenus sur `debruijn-33`

	LR	ED	S-Pitt (2)	LL (2)	SLL (2)	ASLL (2)
Qualité	**3 375 000 000**	*6 749 994 074*	5 495 681 004	*6 749 999 997*	*6 749 999 993*	*6 749 999 997*
Complexité	*13 499 835 000*	**6 747 686 943**	11 642 600 779	12 374 870 094	12 374 907 467	10 125 181 355
User time	2 590,70	2 749,56	2 946,8	3 014,69	5 477	4 686,13
System time	1 176,61	1 386,98	1 251,69	1 280,43	6 814,66	6 206,04
Total	≈ 1H03	≈ 1H09	≈ 1H10	≈ 1H12	≈ 3H25	≈ 3H02

TAB. 4.13 – Récapitulatif des résultats obtenus sur `grid-75000.90000`

	LR	ED	S-Pitt (3)	LL (3)	SLL	ASLL
Qualité	**536 870 912**	1 073 697 074	1 071 341 001	*1 073 741 806*	*1 073 741 823*	*1 073 741 823*
Complexité	*16 106 127 360*	**1 466 269 232**	2 934 632 479	2 161 328 737	2 147 516 642	2 147 434 650
User time	2 026,02	1 193,57	1 401,79	1 180,93	1 326,11	1 295,89
System time	439,95	807,89	761,02	817,92	2 739,69	2 762,33
Total	≈ 42 min	≈ 34 min	≈ 36 min	≈ 33 min	≈ 1H08	≈ 1H08

TAB. 4.14 – Récapitulatif des résultats obtenus sur `hypercube-30`

	LR	ED	S-Pitt (5)	LL	SLL	ASLL
Qualité	*500 000*	70 000	70 146	**35 000**	**35 000**	*500 000*
Complexité	*17 500 000 000*	16 275 036 279	**16 269 962 662**	*17 500 035 000*	*17 500 035 000*	*17 500 500 000*
User time	1 275,82	1 244,55	1 253,63	1 241,02	6 264,24	5 914,35
System time	119,01	98,89	94,53	91,70	16 046,29	16 408,64
Total	≈ 24 min	≈ 23 min	≈ 22 min	≈ 23 min	≈ 6H12	≈ 6H12

TAB. 4.15 – Récapitulatif des résultats obtenus sur `compbip-35000.500000`

	LR	ED	S-Pitt (5)	LL	SLL	ASLL
Qualité	*249 999*	120 478	120 042	**70 000**	**70 000**	*249 999*
Complexité	**249 999**	9 066 622 171	9 097 222 900	12 600 081 857	12 600 081 857	3 097 856
User time	1,62	846,32	864,12	901,97	4 472,10	4,52
System time	15,71	74,32	72,13	69,06	11 587,21	24,34
Total	17,33 sec	≈ 16 min	≈ 16 min	≈ 17 min	≈ 4H28	28,86 sec

TAB. 4.16 – Récapitulatif des résultats obtenus sur `spedens-70000.180000`

Résultats obtenus au quatrième niveau

Dans ce niveau, nous avons repoussé au maximum les limites (en terme de mémoire et d'espace disque) de notre matériel. Nous avons généré deux instances : un graphe peu dense et un graphe dense. Le tableau 4.17 fournit les caractéristiques de ces deux graphes.

L'instance `butterfly-30` prend 1,17 To sur le disque dur (248 Go pour le fichier `.deg` et 960 Go pour le fichier `.list`). Quant à l'instance `compbip-250000.380000`, elle prend 1,41 To sur le disque dur (4,81 Mo pour le fichier `.deg` et 1,41 To pour le fichier `.list`).

Le tableau 4.18 regroupe les résultats obtenus sur `butterfly-30`. Sur ce graphe, nous n'avons pu exécuter que l'algorithme LL. En effet, notre machine (qui dispose de 4 Go de mémoire vive) ne peut pas allouer en mémoire un tableau de 33 milliards de bits. Nous avons donc exécuté une fois LL, en appliquant une permutation circulaire aléatoire sur les labels.

Le tableau 4.19 regroupe les résultats obtenus sur `compbip-250000.380000`. Sur ce graphe, nous avons pu exécuter les algorithmes LR, ED, S-Pitt, LL, SLL-M et ASLL-M (la taille du fichier `.deg` est inférieure à 5 Mo). Nous les avons donc exécutés une fois chacun, en appliquant une permutation circulaire aléatoire sur les labels pour les trois derniers.

Instance	n	m	$\frac{n}{m}$ (densité)	OPT	Temps de création
`butterfly-30`	33 285 996 544	64 424 509 440	1,935	16 106 127 360	$\approx$ 5H10
`compbip-250000.380000`	630 000	95 000 000 000	150 793	250 000	$\approx$ 6H32

TAB. 4.17 – Caractéristiques des graphes générés pour le quatrième niveau

Qualité	Complexité	*User time*	*System time*	Total
32 707 909 573	*65 828 666 108*	14 541,50	6 321,78	$\approx$ 5H48

TAB. 4.18 – Valeurs obtenues pour l'exécution de l'algorithme LL sur `butterfly-30`

	LR	ED	S-Pitt	LL	SLL-M	ASLL-M
Qualité	380 000	500 000	500 601	531 663	**250 000**	380 000
Complexité	*95 000 000 000*	32 500 406 341	32 350 566 643	**24 585 499 800**	*95 000 250 000*	*95 000 380 000*
User time	6 899,68	3 380,01	3 389,26	1 764,57	7 137,02	7 438,45
System time	558,85	298,47	291,43	172,71	499,48	506,51
Total	$\approx$ 2H04	$\approx$ 1H01	$\approx$ 1H02	$\approx$ 33 min	$\approx$ 2H08	$\approx$ 2H13

TAB. 4.19 – Récapitulatif des résultats obtenus sur `compbip-250000.380000`

Nous allons maintenant analyser et interpréter les différents résultats obtenus, dans un premier temps critère par critère, puis nous tenterons par la suite d'en dégager une synthèse générale.

4.2.2 Analyse de la qualité des solutions construites

Le tableau 4.20 compare de façon relative les performances des six algorithmes obtenues en terme de qualité de solution sur les trente-deux instances que nous avons construites (et qui sont décrites

dans les tableaux 4.1, 4.2, 4.6, 4.7 et 4.10). Nous ne considérons par les deux instances du quatrième niveau, qui étaient surtout destinées à tester nos algorithmes lorsque les limites techniques de notre matériel étaient atteintes.

Rangs / **Algorithmes**	1er	2ème	3ème	4ème	5ème	6ème	*Rang moyen*
LR	26	–	1	–	5	–	*1,69*
ED	–	–	8	6	13	5	*4,47*
S-Pitt	–	10	10	10	2	–	*3,13*
LL	6	2	8	10	5	1	*3,28*
SLL	6	14	5	4	3	–	*2,5*
ASLL	–	–	1	1	14	16	*5,41*

Classement global : LR < SLL < S-Pitt < LL < ED < ASLL

TAB. 4.20 – Performances relatives des six algorithmes en terme de qualité de solution

L'algorithme LR est le grand vainqueur sur ce critère. En effet, non seulement il est premier sur la plupart des instances, mais en plus, il retourne assez souvent la solution optimale. En revanche, lorsqu'il n'est pas premier, il s'avère être mauvais (sur les graphes bipartis complets) voire très mauvais (sur les *SpecialDense*).

L'algorithme SLL offre de bonnes performances sur la plupart des graphes. Il est le meilleur sur les graphes bipartis complets et sur les *SpecialDense* et ses performances ne sont pas très éloignées de LR sur les graphes aléatoires en loi de puissance. En effet, ces trois familles de graphes s'adaptent bien aux algorithmes qui choisissent en priorité les sommets de plus fort degré. Aussi, lorsque l'on a la possibilité de l'exécuter plusieurs fois (c'est ce que nous avons fait de manière systématique pour les deux premiers niveaux), il est troisième sur les *ButterFly* et sur les graphes de *de Bruijn*. En revanche, il est plutôt mauvais sur les grilles et sur les hypercubes (puisque ce sont des graphes où la plupart des sommets sont de même degré et où les sommets restants ont un degré plus petit).

L'algorithme S-Pitt est plutôt bon sur les *ButterFly*, les hypercubes et les grilles (il est souvent deuxième sur ces graphes). Il peut aussi être deuxième sur les graphes de *de Bruijn*, lorsque les algorithmes LL et SLL ne sont pas exécutés plusieurs fois. En revanche, il est moins bon sur les graphes bipartis complets, les *SpecialDense* et sur les graphes aléatoires en loi de puissance. Nous avons remarqué que, contrairement à LL et SLL, le fait de l'exécuter plusieurs fois ne permettait pas souvent d'améliorer ses résultats (et lorsque c'était le cas, on ne gagnait pas grand chose).

L'algorithme LL arrive à être premier sur les graphes bipartis complets et sur les *SpecialDense*. De plus, si on peut l'exécuter plusieurs fois (ce qui est plus simple à réaliser à grande échelle que pour SLL), il peut être deuxième sur les graphes de *de Bruijn*. En revanche, il est moyen sur la plupart des autres graphes. Il peut même être mauvais sur les grilles. Enfin, il est assez imprévisible sur les graphes aléatoires en loi de puissance.

L'algorithme ED est mauvais sur la plupart des graphes. De plus, il atteint son rapport d'ap-

proximation (égal à 2) assez souvent. Cela peut lui permettre dans de rares cas (sur les graphes bipartis complets et sur les *SpecialDense*) de se retrouver devant d'autres algorithmes dont les performances ne sont pas garanties par un rapport de 2.

L'algorithme ASLL est mauvais sur l'ensemble des graphes. Il est même très mauvais sur les graphes aléatoires en loi de puissance (puisqu'il choisit en priorité les sommets de plus faible degré).

L'algorithme LR est donc le meilleur en terme de qualité de solution, suivi de près par l'algorithme SLL, qui se comporte bien sur des graphes en loi de puissance et qui peut retourner de bons résultats (surtout si on l'exécute à plusieurs reprises). On retrouve ensuite l'algorithme S-Pitt qui, même s'il n'est jamais le meilleur, ne retourne jamais dans l'absolu de mauvaise solution.

L'algorithme ED, même s'il possède un meilleur rapport d'approximation que les autres (l'algorithme S-Pitt est 2-approché en moyenne, mais pas en pire cas), est globalement plus mauvais (ce résultat ne fait que confirmer ceux obtenus par *F. Delbot* dans [46] et dans [48]). Quant à l'algorithme ASLL, bien qu'il existe des familles de graphes (assez spécifiques) sur lesquelles il a un bon comportement, comme le montrent ces résultats, il est mauvais sur la plupart des autres graphes.

L'algorithme LL (dont les performances globales ne sont pas très éloignées de S-Pitt) est celui dont les résultats varient le plus : il peut très bien « imiter » le comportement des meilleurs comme celui des plus mauvais. Il gagnerait sans doute à être exécuté à plusieurs reprises, en appliquant à chaque fois une véritable permutation aléatoire sur les labels.

4.2.3 Analyse de la complexité en nombre de requêtes

Le tableau 4.21 compare de façon relative les performances des six algorithmes obtenues en terme de complexité en nombre de requêtes sur les trente-deux instances que nous avons construites pour les trois premiers niveaux.

Rangs / **Algorithmes**	1^{er}	$2^{ème}$	$3^{ème}$	$4^{ème}$	$5^{ème}$	$6^{ème}$	*Rang moyen*
LR	3	7	12	4	–	6	*3,28*
ED	22	7	2	1	–	–	*1,44*
S-Pitt	1	3	11	8	9	–	*3,66*
LL	4	2	1	12	9	4	*4*
SLL	–	1	1	5	12	13	*5,09*
ASLL	2	12	5	5	2	6	*3,34*

Classement global : ED < LR < ASLL < S-Pitt < LL < SLL

TAB. 4.21 – Performances relatives des six algorithmes en nombre de requêtes

L'algorithme ED est premier sur la plupart des graphes. Il est deuxième sur certains graphes aléatoires en loi de puissance. Il est moyen sur les *SpecialDense*. Dans l'ensemble, il devance de manière assez nette les autres algorithmes.

L'algorithme LR effectue très peu de requêtes sur les *SpecialDense*. Il peut être bon sur les graphes aléatoires en loi de puissance. Sur les autres graphes, il atteint assez souvent m requêtes (il ne peut pas faire pire). Même si cela le rend mauvais vis-à-vis des autres algorithmes sur les hypercubes et sur les grilles, il arrive à se maintenir au-dessus sur les autres graphes, notamment parce que LL, SLL et ASLL peuvent effectuer plus de m requêtes.

L'algorithme ASLL est deuxième sur les *ButterFly*, les grilles et les *SpecialDense*. Il est moyen sur les hypercubes. Sur les graphes de *de Bruijn*, il peut effectuer plus de m requêtes. Il est toujours dernier sur les graphes bipartis complets (et il effectue plus de m requêtes dessus). Il est en revanche assez imprévisible sur les graphes aléatoires en loi de puissance.

L'algorithme S-Pitt est mauvais sur les hypercubes (mais moins que LR). Il est moyen sur les autres graphes. Il peut toutefois être bon sur les graphes bipartis complets.

L'algorithme LL est mauvais sur la plupart des graphes. Il peut en revanche être bon sur les hypercubes. Quant aux graphes aléatoires en loi de puissance, comme ASLL, il est assez imprévisible.

L'algorithme SLL est mauvais sur l'ensemble des graphes. Il est même très mauvais sur les graphes aléatoires en loi de puissance. Il peut cependant parfois être bon sur les hypercubes.

L'algorithme ED est donc le meilleur en terme de complexité en nombre de requêtes. Il est suivi par l'algorithme LR, qui doit sa deuxième place en grande partie au fait que les algorithmes LL, SLL et ASLL peuvent effectuer (contrairement à lui) plus de m requêtes. Il est suivi de près par l'algorithme ASLL, qui effectue moins souvent plus de m requêtes que LL et SLL et qui est bon sur certains graphes (les grilles et les *ButterFly* par exemple).

Dans l'absolu, l'algorithme S-Pitt est globalement moyen sur l'ensemble des graphes. Il n'atteint et ne dépasse jamais m requêtes.

Les algorithmes LL et SLL effectuent assez souvent plus de m requêtes. Mais contrairement à SLL, LL peut parfois être bon sur les graphes aléatoires en loi de puissance.

4.2.4 Analyse des temps d'exécution

Nous nous sommes focalisés sur les durées d'exécution des algorithmes au troisième et au quatrième niveau (il est en effet difficile de dégager de réelles différences sur les deux premiers niveaux, où ces durées sont similaires).

Comme nous l'avons précisé dans la section 4.1.4, les temps d'exécution sont exprimés en *user + system time*. Pour obtenir une estimation des temps réels écoulés, il faut multiplier en moyenne par 2,5 les temps de création fournis par les tableaux 4.10 et 4.17. De même, pour les algorithmes LR, ED, S-Pitt, LL, SLL-M et ASLL-M (resp. SLL et ASLL), il faut multiplier en moyenne par 3,2 (resp. 1,6) les durées indiquées dans les différents tableaux.

Les algorithmes SLL et ASLL se distinguent des autres algorithmes puisqu'ils utilisent une tête de lecture supplémentaire pour pouvoir récupérer les degrés des voisins. Leurs temps d'exécution sont donc difficilement comparables aux autres. Nous allons donc nous intéressés principalement aux algorithmes LR, ED, S-Pitt et LL.

De façon générale, on peut corréler le nombre de requêtes au *system time* (on récupère les voisins sur le disque externe), mais aussi au *user time*, puisque chaque requête fait l'objet d'un traitement sur le processeur de la machine (par exemple, une comparaison entre deux labels pour LL ou un test d'appartenance à la mémoire pour ED). En revanche, la taille des solutions joue essentiellement sur le *system time*, puisque l'on se contente d'écrire sur le disque externe (après avoir effectuer un traitement suite à une requête). On peut d'ailleurs observer sur l'instance `hypercube-30` (voir le tableau 4.14 page 118) que l'algorithme ED, qui écrit deux fois plus de sommets que l'algorithme LR, a un *system time* deux fois plus grand.

Cependant, ces deux paramètres ne suffisent pas à interpréter les temps d'exécution. D'autres aspects techniques, liés notamment aux principes de fonctionnement des systèmes d'exploitation (voir par exemple [25]), entrent en jeu. Par exemple, l'accès au disque externe n'est pas direct : pour effectuer ses entrées/sorties, la machine de traitement utilise des tampons. Ainsi, même si l'on récupère uniquement un voisin dans le fichier `.list`, le système en charge en réalité un certain nombre (cette quantité est liée entre autres à la taille des secteurs sur le disque). Par conséquent, le nombre d'accès physiques au disque externe est, la plupart du temps, inférieur au nombre de requêtes effectuées[5]. De ce fait, on ne peut pas identifier de manière certaine une relation entre les temps d'exécution des algorithmes, le nombre de requêtes qu'ils effectuent et le nombre de sommets qu'ils écrivent. En effet, si l'on observe les résultats regroupés dans le tableau 4.19 page 119, on remarque que l'algorithme LR, qui effectue trois fois plus de requêtes que l'algorithme ED, met (seulement) deux fois plus de temps à s'exécuter. En revanche, LL, qui effectue quatre fois moins de requêtes que LR, met quatre fois moins de temps à s'exécuter.

Sur les graphes peu denses (pour lesquels $n \in \Theta(m)$), les temps d'exécution sont proches. En effet, il y a bien souvent une compensation entre le nombre de sommets retournés et le nombre de requêtes effectuées. De plus, les algorithmes qui produisent les meilleures solutions sont, assez souvent, ceux qui effectuent le plus de requêtes (en tout cas, un algorithme n'est jamais à la fois le meilleur en terme de qualité de solution et en terme de complexité en nombre de requêtes). Par exemple, si l'on regarde le tableau 4.11 page 117, on peut voir que l'algorithme LR, qui effectue deux fois plus de requêtes que l'algorithme ED, met quasiment autant de temps à s'exécuter que lui, puisque ED retourne par ailleurs deux fois plus de sommets.

Le nombre de sommets de la solution (lorsqu'il est important) influence d'autant plus les temps d'exécution puisque, d'un point de vue technique, le fait d'écrire sur le disque externe est, en règle générale, plus long que le fait de lire.

Sur les graphes denses, l'analyse des temps d'exécution est moins complexe car la taille des solutions produites est beaucoup plus faible que le nombre de requêtes effectuées. Par conséquent, les temps d'exécution sont plus influencés par le nombre de requêtes que par la taille des solutions.

Sur l'instance `spedens-70000.180000` (voir le tableau 4.16 page 118), on remarque que les algorithmes LR et ASLL, qui effectuent très peu de requêtes, sont bien plus rapides que les autres algorithmes (et ce malgré le fait que ASLL effectue des requêtes de nature différente).

[5] On rejoint dans ce sens les considérations d'ordre pratique mises en avant dans l'approche I/O-efficient.

Pour les algorithmes SLL et ASLL, pour qui les requêtes nécessitent plus de calculs et donc plus de temps, on peut dire que le nombre de requêtes effectuées a une influence plus forte sur les temps d'exécution par rapport au nombre de sommets retournés. Par exemple, si l'on reprend le tableau 4.11 page 117, on constate que SLL effectue plus de requêtes que ASLL et que ses durées sont plus élevées, malgré le fait qu'il écrive moins de sommets dans la solution.

4.3 Synthèse générale

Nous avons implémenté six des algorithmes que nous avons étudiés lors des précédents chapitres et nous les avons testés sur plusieurs types de graphes. Nous avons défini pour cela plusieurs niveaux de tailles (en fonction de critères relatifs à notre machine de traitement). Nous avons obtenu dans l'ensemble des temps d'exécution raisonnables (moins d'une journée), et ce même pour des graphes constitués de plusieurs dizaines de milliards d'arêtes et de sommets. Ces algorithmes sont donc bien adaptés pour le traitement de grandes instances.

Nous avons vu qu'en terme de qualité de solution, l'algorithme LR était globalement le meilleur, suivi de près par l'algorithme SLL, tandis que les algorithmes ED et ASLL étaient les plus mauvais. Par ailleurs, nous avons vu qu'en terme de complexité en nombre de requêtes, que c'était l'algorithme ED qui était le meilleur, tandis que l'algorithme SLL était le plus mauvais. Par conséquent, il est difficile de choisir un algorithme qui satisfasse en même temps ces deux critères.

En effet, nous avons remarqué que, sur les graphes peu denses, ces deux critères pouvaient influencer les temps d'exécution, qui étaient souvent proches. Il est donc préférable, pour ces graphes, de privilégier la qualité de la solution au nombre de requêtes. L'algorithme LR est donc un bon candidat. Cependant, il a besoin d'utiliser n bits d'espace mémoire pour s'exécuter, ce qui devient impossible à partir d'une certaine taille.

Pour les graphes denses, on peut toujours créer un tableau de n bits sur notre machine (puisque n n'est pas grand, par rapport à m). Ce n'est donc pas un problème d'exécuter les algorithmes LR, ED et S-Pitt. En revanche, on peut avoir des temps d'exécution différents, ce qui rend le choix d'un algorithme bien délicat, puisque ceux qui produisent les meilleures solutions sont, en règle générale, ceux qui mettent le plus de temps à s'exécuter.

Malgré ses performances moyennes (voires mauvaises) en terme de qualité de solution et de complexité en nombre de requêtes, l'algorithme LL reste celui qui est le plus adapté aux grandes tailles, puisqu'il n'utilise pas de tableau de n bits et puisqu'il ne récupère pas les degrés des voisins (c'est d'ailleurs le seul algorithme que nous avons pu exécuter sur l'instance `butterfly-30`).

A l'inverse, SLL et ASLL, qui effectuent beaucoup plus d'accès disques que les autres, atteignent rapidement leurs limites. Toutefois, sur des graphes denses, on peut les remplacer par SLL-M et ASLL-M (nous avons même pu le faire sur l'instance `compbip-250000.380000`). Ils peuvent dans ce cas rivaliser avec les autres algorithmes.

Une extension de cette étude expérimentale serait de générer d'autres types de graphes, comme

des arbres. Cela nous permettrait de pouvoir tester l'algorithme OT.

On pourrait aussi effectuer des tests plus poussés sur l'algorithme LL. En générant de véritables permutations aléatoires sur les labels et en l'exécutant de manière parallèle, on obtiendrait peut-être de meilleurs résultats et on améliorerait ses temps d'exécution. On pourrait donc l'exécuter un plus grand nombre de fois sur chaque instance et espérer ainsi obtenir de meilleurs résultats en terme de qualité de solution.

Nous avons vu que le modèle sur lequel nous nous sommes appuyés pour réaliser nos expérimentations présentait quelques différences par rapport au modèle proposé dans la section 1.2. Une perspective serait d'étendre ce modèle théorique pour essayer de mieux prendre en compte les différentes contraintes du système.

Conclusion et perspectives

Dans cette thèse, nous nous sommes penchés sur un problème d'optimisation (le VERTEX COVER) dans un contexte de traitement de grandes instances de données.

Nous avons tout d'abord proposé dans le chapitre 1 un modèle théorique basé sur trois contraintes de traitement. Nous avons montré, à travers la présentation d'autres modèles existants, que ces contraintes faisaient la synthèse de propriétés issues de ces différents modèles.

Nous avons ensuite analysé et comparé dans le chapitre 2 les performances de trois algorithmes *memory-efficient* (ils utilisent une mémoire de taille constante par rapport à la quantité de données traitées) pour ce problème.

Nous avons par la suite étudié dans le chapitre 3 les performances d'autres algorithmes utilisant n bits d'espace mémoire pour s'exécuter.

Enfin, nous avons mené une étude expérimentale sur de gros graphes. Nous en avons présenté les caractéristiques et les résultats dans le chapitre 4.

Résultats obtenus et remarques

Dans le chapitre 1, nous avons prouvé que l'on ne pouvait résoudre de façon optimale et en une seule passe le problème du VERTEX COVER avec moins de n^2 bits mémoire.
Ce résultat (assez négatif) conforte l'idée qu'il est préférable de privilégier, lorsque n est très grand, des solutions approchées, souvent moins gourmandes en mémoire.

Dans le chapitre 2, nous avons donné les formules exactes exprimant les tailles moyennes des solutions retournées ainsi que le nombre moyen de requêtes effectuées par les algorithmes LL, SLL et ASLL. En utilisant ces formules pour les comparer, nous avons exhibé des familles de graphes pour lesquelles, pour ces deux critères, aucun de ces trois algorithmes ne pouvait être désigné de façon absolue comme étant le meilleur.

Dans le chapitre 3, nous avons étudié l'algorithme OT, qui est optimal sur les arbres mais qui a besoin, dans certains cas, d'effectuer plusieurs passes. Nous avons montré que pour tout arbre, il existait un ordre de traitement des sommets pour lequel il n'effectuait qu'une seule passe. Nous avons aussi montré qu'il pouvait effectuer moins de m requêtes mais qu'il pouvait par ailleurs en effectuer de l'ordre de m^2.
Cet algorithme (différent des autres que nous avons étudiés puisque adapté pour une famille de graphe en particulier) montre qu'il est toujours possible de construire en une seule passe une

couverture optimale pour un arbre avec n bits mémoire. De ce fait, il est possible de retourner une solution optimale sur des arbres de grande taille.

Nous avons analysé la complexité en nombre de requêtes des algorithmes LR et ED et nous avons montré qu'ils ne pouvaient pas effectuer plus de m requêtes.

Nous avons aussi étudié un algorithme probabiliste : Pitt. Nous nous sommes intéressés à la qualité des solutions qu'il construisait en fonction de la quantité de mémoire dont il disposait. Nous avons montré que, même en ayant la possibilité de stocker la moitié des sommets en mémoire, il pouvait être très mauvais en moyenne.
Ce résultat (plutôt négatif) montre que le fait de ne pas pouvoir stocker tous les sommets du graphe peut dégrader de façon non négligeable les solutions produites.

Dans le chapitre 4, nous avons vu globalement que les algorithmes que nous avons étudiés étaient adaptés au traitement de grandes instances puisque, sur des graphes de l'ordre de plusieurs dizaines de milliards de sommets et d'arêtes, leurs temps d'exécution étaient raisonnables (moins d'une journée sur une machine de bureau).

Nous avons constaté cependant que les algorithmes SLL et ASLL (qui récupèrent les degrés des voisins) présentaient bien souvent des durées supérieures aux autres algorithmes.

Par ailleurs, nous avons observé qu'en règle générale, il y avait un lien entre le nombre de requêtes effectuées et la taille des solutions construites, mais que ces deux critères ne suffisaient pas à expliquer les temps d'exécution des algorithmes.

De manière générale, les travaux menés durant cette thèse peuvent fournir des indicateurs pour choisir le ou les algorithmes qui conviennent le mieux pour traiter le problème du VERTEX COVER sur de gros graphes.

Nous nous sommes intéressés à deux familles d'algorithmes : ceux qui n'utilisent pratiquement pas de mémoire (les algorithmes *memory-efficient*) et ceux qui utilisent n bits mémoire. Nous avons remarqué de plus dans le chapitre 4 qu'il y avait deux cas distincts à prendre en compte : les graphes denses et les graphes peu denses.

Sur les graphes denses (pour lesquels le nombre de sommets est petit par rapport au nombre d'arêtes), tous les algorithmes que nous avons testés sont, en pratique, adaptés (pour SLL et ASLL, on peut charger en mémoire les valeurs nécessaires pour accélérer le calcul des degrés). Il convient donc de privilégier sur ces graphes l'algorithme LR puisqu'il retourne, en pratique, de bonnes solutions et qu'il ne peut pas effectuer plus de m requêtes.

Sur les graphes peu denses (pour lesquels le nombre de sommets est aussi important que le nombre d'arêtes), les algorithmes qui ont besoin de n bits mémoire atteignent leurs limites (et nous ne pouvons pas exécuter les versions « mémoire » de SLL et de ASLL). Par conséquent, seul l'algorithme LL reste adapté.

Bien entendu, la difficulté est accrue si l'on ne connaît pas à l'avance le nombre de sommets du graphe (c'est d'ailleurs souvent le cas dans le modèle streaming). Si l'on exécute les algorithmes à

n bits mémoire et qu'ils ne peuvent pas stocker tous les sommets, les solutions qu'ils vont produire pourront être très mauvaises.

L'algorithme LL est le seul (parmi ceux que nous avons étudiés) qui, quelle que soit la densité du graphe, peut toujours s'exécuter. Certes, il n'offre pas les mêmes performances en terme de qualité de solution que LR, mais il peut toujours retourner la couverture optimale d'un graphe et, comme l'ont montrées les expériences réalisées par *F. Delbot* (voir [46] et [48]), il est souvent moins mauvais que l'algorithme ED (qui est pourtant 2-approché). De plus, son exécution est parallélisable : on peut réduire ses temps de calcul si l'on dispose de plusieurs machines de traitement. On peut aussi l'exécuter à plusieurs reprises, en effectuant à chaque fois une permutation aléatoire sur les labels des sommets et en conservant la meilleure des solutions produites.

Choisir un algorithme (qui plus est d'approximation) qui soit à la fois performant (en terme de qualité de solution et de complexité) et qui satisfasse les contraintes du modèle que l'on considère est délicat. En effet, les algorithmes les plus performants ne sont pas toujours les mieux adaptés.

Dans les travaux que nous avons réalisés, nous sommes parvenus à la conclusion qu'il est préférable de choisir au départ l'algorithme qui est le mieux adapté au contexte dans lequel on se place plutôt que de choisir celui qui est le plus performant. En effet, nous pensons qu'il est plus simple de modifier le premier pour le rendre plus efficace et plus performant (et sans que cela ne lui fasse perdre ses bonnes propriétés vis-à-vis du modèle que l'on considère) plutôt que de tenter d'adapter le second sans dégrader ses performances. C'est en tout cas ce que nous pensons pouvoir faire pour le problème du VERTEX COVER sur de grands graphes avec l'algorithme LL.

Le fait de traiter des données de grande taille nous oblige à être précis et exigeant dans les analyses que nous menons. Par exemple, même si $2m$ et $3m$ sont de même ordre de grandeur (en $\mathcal{O}(m)$), lorsque m est très grand, la différence entre les deux valeurs peut être considérable.

Idéalement, il faudrait avoir à la fois des garanties de performances sur les algorithmes que nous utilisons (comme, par exemple, savoir que l'algorithme LR ne peut pas effectuer plus de m requêtes quoiqu'il arrive), ainsi que des outils nous permettant de prédire au mieux leurs comportements (comme, par exemple, pouvoir calculer de façon analytique la taille moyenne des solutions retournées par l'algorithme SLL sur une famille de graphes).

Perspectives

Une première idée de poursuite de ces travaux serait de compléter les résultats théoriques obtenus dans les chapitres 2 et 3, en prouvant la conjecture 1 à propos de SLL et de ASLL sur les arbres, en déterminant les formules exactes des variances pour les algorithmes LL, SLL et ASLL ou bien en exprimant de manière exacte le nombre moyen de requêtes (et de passes, pour OT) effectuées par les algorithmes OT, LR, ED et Pitt. Ces différentes valeurs nous permettraient de comparer de façon plus précise les algorithmes que nous avons étudiés.

Une autre idée de poursuite de ces travaux serait de compléter l'étude expérimentale que nous avons menée en implémentant l'algorithme OT et en le testant sur des arbres. Cela nous permettrait de tester ses performances réelles et d'observer, en pratique, s'il est bien adapté pour traiter de grandes instances.

On pourrait aussi effectuer des expérimentations plus poussées sur l'algorithme LL, en utilisant plusieurs machines de traitement pour le paralléliser. Cela permettrait de réduire ses temps de calcul. On pourrait alors l'exécuter à plusieurs reprises et, en générant de véritables permutations aléatoires sur les labels des sommets à chaque fois, on pourrait ainsi espérer améliorer la qualité des solutions construites.

A plus long terme, nous pourrions étudier d'autres problèmes d'optimisation dans ce contexte (réaliste) de traitement de données de grande taille. Nous pourrions aussi y adapter des problèmes polynomiaux (comme, par exemple, la recherche du diamètre dans un graphe), en proposant des méthodes de résolution plus rapides.

Pour conclure, je dirais que le traitement de grandes masses de données est un enjeu majeur qui va sans doute (si ce n'est pas déjà le cas) devenir un axe important voire incontournable de la recherche en informatique. C'est un fait à prendre en compte, je pense, pour les travaux futurs.

Bibliographie

[1] Matti Åstrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. A Local 2-approximation Algorithm for the Vertex Cover Problem. In *DISC '09 : Proceedings of the 23rd International Conference on Distributed Computing*, pages 191–205. Springer-Verlag Berlin, Heidelberg, 2009.

[2] James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook. A Functional Approach to External Graph Algorithms. *Algorithmica*, 32(3) :437–458, 2002.

[3] Faisal N. Abu-Khzam, Michael A. Langston, Pushkar Shanbhag, and Christopher T. Symons. Scalable Parallel Algorithms for FPT Problems. *Algorithmica*, 45(3) :269–284, 2006.

[4] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9) :1116–1127, 1988.

[5] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the Streaming Model Augmented with a Sorting Primitive. In *FOCS '04 : Proccedings of the 45th Annual IEEE Symposium on Fundations of Computer Science*, pages 540–549, Rome, Italie, 2004.

[6] Kook Jin Ahn and Sudipto Guha. Linear Programming in the Semi-Streaming Model with Application to the Maximum Matching Problem. *CoRR*, abs/1104.2315, 2011.

[7] Deepak Ajwani. Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Allemagne, 2005.

[8] Susanne Albers. Online Algorithms : a Survey. *Mathematical Programming*, 97(1–2) :3–26, 2003.

[9] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Biologie Moléculaire de la Cellule, deuxième édition*. Médecine-Sciences, Flammarion, 1989.

[10] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences*, 58(1) :137–147, 1999. Présenté à STOC en 1996.

[11] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. Wiley, New York, 1992.

[12] Eric Angel, Romain Campigotto, and Christian Laforest. Algorithms for the Vertex Cover Problem on Large Graphs. Technical Report No 1, IBISC – Université d'Evry, 2010.

[13] Eric Angel, Romain Campigotto, and Christian Laforest. Comparaison d'algorithmes pour le problème du Vertex Cover sur de grands graphes (résumé de 2 pages). In *ROADEF*, Université Toulouse 1 Capitole, 2010.

[14] Eric Angel, Romain Campigotto, and Christian Laforest. Analysis and Comparison of Three Algorithms for the Vertex Cover Problem on Large Graphs with Low Memory Capacities. *Algorithmic Operations Research*, 6(1) :56–67, 2011.

[15] Lars Arge, Ulrich Meyer, Laura Toma, and Norbert Zeh. On External-Memory Planar Depth-First Search. *Journal of Graph Algorithms and Applications*, 7(2) :105–129, 2003.

[16] Lars Arge and Norbert Zeh. I/O-Efficient Strong Connectivity and Depth-First Search for Directed Planar Graphs. In *FOCS '03 : Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 261–270. IEEE Computer Society, 2003.

[17] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof Verification and the Hardness of Approximation Problems. *Journal of the ACM*, 45(3) :501–555, 1998.

[18] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs : A new characterization of NP. *Journal of the ACM*, 45(1) :70–122, 1998.

[19] Eyjolfur Asgeirsson and Cliff Stein. Vertex Cover Approximation on Random Graphs. In C. Demetrescu, editor, *WEA 2007*, volume LNCS 4525, pages 285–296. Springer-Verlag Berlin, Heidelberg, 2007.

[20] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation*. Springer, 1999.

[21] David Avis and Tomokazu Imamura. A List Heuristic for Vertex Cover. *Operations Research Letters*, 35 :201–204, 2006.

[22] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *PODS '02 : Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, 2002.

[23] Reuven Bar-Yehuda and Shimon Even. A Local Ratio Theorem for Approximating the Weighted Vertex Cover Problem. *Annals of Discrete Mathematics*, 25 :27–46, 1985.

[24] Reuven Bar-Yehuda, Danny Hermelin, and Dror Rawitz. Minimum Vertex Cover in Rectangle Graphs. In *ESA '10 : Proceedings of the 18th Annual European Conference on Algorithms, Part I*, pages 255–266. Springer-Verlag, 2010.

[25] Joffroy Beauquier and Béatrice Bérard. *Systèmes d'exploitation : concepts et algorithmes*. Ediscience, 2001.

[26] Babak Behsaz, Pooya Hatami, and Ebadollah S. Mahmoodian. On Minimum Vertex Cover of Generalized Petersen Graphs. *Australasian Journal of Combinatorics*, 40 :253–264, 2008.

[27] Etienne Birmelé, François Delbot, and Christian Laforest. Mean Analysis of an Online Algorithm for the Vertex Cover Problem. *Information Processing Letters*, 109 :436–439, 2009.

[28] Tom Bohman, Alan Frieze, Miklós Ruszinkó, and Lubos Thoma. Vertex Covers by Edge Disjoint Cliques. *Combinatorica*, 21(2) :171–197, 2001.

[29] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, USA, 1998.

[30] Boštjan Brešar, František Kardoš, Jàn Katrenič, and Gabriel Semanišin. Minimum k-path Vertex Cover. *Discrete Applied Mathematics*, 159(12) :1189–1195, 2011.

[31] Jean Cardinal, Marek Karpinski, Richard Schmied, and Claus Viehmann. Approximating Vertex Cover in Dense Hypergraphs. *CoRR*, abs/1012.2573, 2010.

[32] Jean Cardinal, Martine Labbé, Stefan Langerman, Eythan Levy, and Hadrien Mélot. A Tight Analysis of the Maximal Matching Heuristic. In *COCOON '05 : Proceedings of the 11th Annual International Computing and Combinatorics Conference*, volume LNCS 3595, pages 701–709, Kunming, China, 2005. Springer-Verlag.

[33] Yair Caro. New Results on the Independence Number. Technical report, Tel Haviv University, 1979.

[34] Dustin A. Cartwright, Maria Angélica Cueto, and Enrique A. Tobis. The Maximum Independent Sets of de Bruijn Graphs of Diameter 3. *Electronic Journal of Combinatorics*, 2010.

[35] F. H. Chang, Hong-Lin Fu, Frank K. Hwang, and B. C. Lin. The Minimum Number of e-Vertex-Covers among Hypergraphs with e Edges of Given Ranks. *Discrete Applied Mathematics*, 157(1) :164–169, 2009.

[36] Yeim-Kuan Chang and Yung-Chieh Lin. A Fast and Memory Efficient Dynamic IP Lookup Algorithm Based on B-Tree. In *International Conference on Advanced Information Networking and Applications*. IEEE, 2009.

[37] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better Streaming Algorithms for Clustering Problems. In *STOC '03 : Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 30–39, San Diego, California, 2003.

[38] Eric Y. Chen. Geometric Streaming Algorithms with a Sorting Primitive. In T. Tokuyama, editor, *ISAAC '07 : Proceedings of the 18th International Symposium on Algorithms and Computation*, volume LNCS 4835, pages 512–524, Sendai, Japan, 2007. Springer-Verlag.

[39] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex Cover : Further Observations and Further Improvements. *Journal of Algorithms*, 41(2) :280–301, 2001.

[40] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved Parameterized Upper Bounds for Vertex Cover. In *MFCS '06 : Proceedings of the 31st International Symposium of Mathematical Foundations of Computer Science*, volume LNCS 4162, pages 238–249, Stará Lesná, Slovaquie, 2006.

[41] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *STOC '71 : Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, USA, 1971.

[42] Geoffrey M. Cooper and Robert E. Hausman. *The Cell : a Molecular Approach, fifth edition.* ASM Press, Washington, D.C., 2009.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, third edition.* MIT Press, 2009.

[44] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In *Formal Methods in System Design*, volume 1, pages 275–288, 1992.

[45] Nicolaas Govert de Bruijn. A Combinatorial Problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49 :758–764, 1946.

[46] François Delbot. *Au delà de l'évaluation en pire cas : comparaison et évaluation en moyenne de processus d'optimisation pour le problème du vertex cover et des arbres de connexion de groupes dynamiques.* PhD thesis, Université d'Evry, 2009.

[47] François Delbot and Christian Laforest. A Better List Heuristic for Vertex Cover. *Information Processing Letters*, 107 :125–127, 2008.

[48] François Delbot and Christian Laforest. Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover. *ACM Journal of Experimental Algorithmics*, 2011. Accepté.

[49] Marc Demange and Vangelis Th. Paschos. On-line Vertex-Covering. *Theoretical Computer Science*, 332 :83–108, 2005.

[50] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL : Standard Template Library for XXL Data Sets. *Software : Practice and Experience*, 38(6) :539–637, 2008. Librairie disponible sur `http://stxxl.sourceforge.net`.

[51] Camil Demetrescu, Bruno Escoffier, Gabriel Moruz, and Andrea Ribichini. Adapting Parallel Algorithms to the W-Stream Model, with Applications to Graph Problems. *Theoretical Computer Science*, 411(44–46) :3994–4004, 2010.

[52] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading Off Space for Passes in Graph Streaming Problems. In *SODA '06 : Proceddings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 714–723, Miami, Florida, 2006.

[53] Irit Dinur and Samuel Safra. On the Hardness of Approximating Minimum Vertex Cover. *Annals of Mathematics*, 162(1) :439–485, 2005.

[54] Rod G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer, 1999.

[55] Philip M. Duxbury and C. W. Fay IV. Precise Polynomial Heuristic for an NP-Complete Problem, 2003. Disponible sur `http://arxiv.org/abs/cond-mat/0309324v1`.

[56] Sebastian Eggert, Lasse Kliemann, and Anand Srivastav. Bipartite Graph Matchings in the Semi-Streaming Model. In *ESA '09 : Proceedings of the 17th Annual European Symposium on Algorithms*, pages 492–503, 2009.

[57] Bruno Escoffier, Laurent Gourvès, and Jérôme Monnot. Complexity and Approximation Results for the Connected Vertex Cover Problem in Graphs and Hypergraphs. *Journal of Discrete Algorithms*, 8 :36–49, 2010.

[58] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On Graph Problems in a Semi-Streaming Model. *Theoretical Computer Science*, 348(2–3) :207–216, 2005.

[59] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph Distances in the Data-Stream Model. *SIAM Journal on Computing*, 38(5) :1709–1727, 2008. Présenté à SODA en 2005.

[60] Joan Feigenbaum, Sampath Kannan, and Jian Zhang. Computing Diameter in the Streaming and Sliding-Window Models. *Algorithmica*, 41(1) :25–41, 2002.

[61] Philippe Flajolet and G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2) :182–209, 1985.

[62] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[63] Pierre Fouilhoux and A. Ridha Mahjoub. Solving VLSI Design and DNA Sequencing Problems using Bipartization of Graphs. *Computational Optimization and Applications*, pages 1–33, 2010.

[64] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, 1979.

[65] Michael R. Garey, David S. Johnson, and Larry Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, 1(3) :237–267, 1976.

[66] Maurice Glaymann. Où le premier n'est pas toujours premier... *Educational Studies in Mathematics*, 7(1-2) :83–88, 1976.

[67] Oded Goldreich. Property Testing in Massive Graphs. In James Abello, Panos M. Pardalos, and Mauricio G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 123–147. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[68] Oded Goldreich, editor. *Property Testing : Current Research and Surveys*, volume LNCS 6390. Springer-Verlag, 2010.

[69] Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property Testing and Its Connection to Learning and Approximation. *Journal of the ACM*, 45(4) :653–750, 1998.

[70] Matteo Golfarelli and Stefano Rizzi. *Data Warehouse Design : Modern Principles and Methodologies*. McGraw-Hill, 2009.

[71] Irwin John Good. Normal Recurring Decimals. *Journal of the London Mathematical Society*, 21(3) :167–169, 1946.

[72] Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. Distributed Weighted Vertex Cover via Maximal Matchings. *ACM Transaction on Algorithms*, 5(1), 2008.

[73] Venkatesan Guruswami and Ryan O'Donnel. PCP Course Notes, University of Washington, 2005. Disponible sur `http://www.cs.washington.edu/education/courses/cse533/05au`.

[74] Philip Hall. On Representatives of Subsets. *Journal of the London Mathematical Society*, 10(1) :26–30, 1935.

[75] Bjarni V. Halldórsson, Magnús M. Halldórsson, Elena Losievskaja, and Mario Szegedy. Streaming Algorithms for Independent Sets. In *ICALP '10 : Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming*, pages 641–652. Springer-Verlag, 2010.

[76] Alexander K. Hartmann and Martin Weigt. Statistical Mechanics of the Vertex-Cover Problem. *Journal of Physics A : Mathematical and General*, 36(43), 2003.

[77] Monika Rauch Heizinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on Data Streams. Technical report, Digital System Search Center, Palo Alto, California, 1998.

[78] John Craig Venter *et al.* The Sequence of the Human Genome. *Science*, 291 :1304–1351, 2001.

[79] George Karakostas. A Better Approximation Ratio for the Vertex Cover Problem. *ACM Transactions on Algorithms (TALG)*, 5(4), 2009.

[80] Richard M. Karp. Reducibility among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[81] Dénes Kőnig. Gràfok és màtrixok. *Matematikai és Fizikai Lapok*, 31 :116–119, 1931.

[82] Subhash Khot and Oded Regev. Vertex Cover Might be Hard to Approximate to Within $2-\epsilon$. *Journal of Computer and System Sciences*, 74(3) :335–349, 2008.

[83] Christos Koufogiannakis and Neal E. Young. Distributed and Parallel Algorithms for Weighted Vertex Cover and other Covering Problems. In *PODC '09 : Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pages 171–179. ACM, 2009.

[84] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[85] Giuseppe Lancia, Vineet Bafna, Sorin Istrail, Ross Lippert, and Russel Schwartz. SNPs Problems, Complexity and Algorithms. In F. Meyer auf der Heide, editor, *ESA 2001*, volume LNCS 2161, pages 182–193. Springer-Verlag Berlin Heidelberg, 2001.

[86] Hoon-Jae Lee, Han-Soo Kim, and Ju-Wook Jang. A New Approximated VC Scheme to Prevent D-DoS Attack against Networks of ASes. In *International Joint Conference on Information and Communication*, 2004.

[87] Leonid Levin. Universal'nye Perebornye Zadachi. *Problemy Peredachi Informatsii*, 9(3) :265–266, 1973. English translation, "Universal Search Problems", in B. A. Trakhtenbrot (1984).

[88] Nicolas Lichiardopol. Independence Number of de Bruijn Graphs. *Discrete Mathematics*, 306(12) :1145–1160, 2006.

[89] Ross Lippert, Russel Schwartz, Giuseppe Lancia, and Sorin Istrail. Algorithmic Strategies for the Single Nucleotide Polymorphism Haplotype Assembly Problem. *Brefings in Bioinformatics*, 3(1) :23–31, 2002.

[90] Ulrich Meyer. On Trade-Offs in External-Memory Diameter-Approximation. In *SWAT '08 : Proceedings of the 11th Scandinavian Workshop on Algorithm Theory*, volume LNCS 5124, pages 426–436, Gothenburg, Suède, 2008. Springer-Verlag.

[91] Ulrich Meyer. *Efficient Algorithms*, chapter Via Detours to I/O-Efficient Shortest Paths, pages 219–232. Springer-Verlag, 2009.

[92] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume LNCS 2625. Springer-Verlag, 2003.

[93] Michael Molloy and Bruce Reed. A Critical Point for Random Graphs With a Given Degree Sequence. *Random Structures and Algorithms*, pages 161–179, 1995.

[94] Burkhard Monien and Erwald Speckenmeyer. Ramsey Numbers and an Approximation Algorithm for the Vertex Cover Problem. *Acta Informatica*, 22(1) :115–123, 1985.

[95] Hannes Moser. Exact Algorithms for Generalizations of Vertex Cover. Master's thesis, Friedrich Schiller Universität at Jena, 2005.

[96] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, 1995.

[97] S. Muthu Muthukrishnan. Data Streams : Algorithms and Applications, 2003. Disponible sur `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[98] S. Muthu Muthukrishnan and Andrew McGregor. Data Streams Algorithms, 2009. Lecture Notes from Data Streams course at Barbados.

[99] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[100] Thomas C. O'Connell. *Fundamental Problems in Computing*, chapter A Survey of Graph Algorithms under Extended Streaming Models of Computation, pages 455–476. Springer Science + Business Media, 2009.

[101] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, Approximation and Complexity Classes. *Journal of Computer and System Sciences*, 43(3) :425–440, 1991.

[102] Michal Parnas and Dana Ron. Approximating the Minimum Vertex Cover in Sublinear Time and a Connection to Distributed Algorithms. *Theoretical Computer Science*, 381 :183–196, 2007.

[103] Vangelis Th. Paschos. *Complexité et approximation polynomiale*. Hermès Science, 2004.

[104] Stéphane Pérennes and Ignasi Sau Valls. Sur la Conjecture des Jeux Uniques. Technical Report 6691, INRIA Sophia-Antipolis, 2008.

[105] Shariefuddin Pirzada and Ashay Dharwadker. Applications of Graph Theory. *Journal of The Korean Society for Industrial and Applied Mathematics (KSIAM)*, 11(4) :19–38, 2007.

[106] Leonard Pitt. A Simple Probabilistic Approximation Algorithm for Vertex Cover. Technical Report 404, Yale University, Department of Computer Science, 1985.

[107] Igor Razgon. Faster Computation of Maximum Independent Set and Parameterized Vertex Cover for Graphs with Maximum Degree 3. *Journal of Discrete Algorithms*, 7 :191–212, 2009.

[108] Silvia Richter, Malte Helmert, and Charles Gretton. A Stochastic Local Search Approach to Vertex Cover. In M. Beetz J. Hertzberg and R. Englert, editors, *KI '07 : Proceedings of the 30th German Conference on Artificial Intelligence*, pages 412–426. Springer-Verlag Berlin Heidelberg, 2007.

[109] Dana Ron. Algorithmic and Analysis Techniques in Property Testing. *Foundations and Trends in Theoretical Computer Science*, 5(2) :73–205, 2010.

[110] Chantal Roth-Korostensky. *Algorithms for Building Multiple Sequence Alignments and Evolutionary Trees*. PhD thesis, ETH Zurich, Institute of Scientific Computing, 2000.

[111] Ronitt Rubinfeld and Madhu Sudan. Robust Characterization of Polynomials with Applications to Program Testing. *SIAM Journal on Computing*, 25(2) :252–271, 1996.

[112] Matthias Ruhl. *Efficient Algorithms for New Computationnal Models*. PhD thesis, Department of Computer Science, MIT, Cambridge, Massachusetts, 2003.

[113] Carla Savage. Depth-First Search and the Vertex Cover Problem. *Information Processing Letters*, 14(5) :233–235, 1982.

[114] Russel Schwartz. Theory and Algorithms for the Haplotype Assembly Problem. *Communications in Information and Systems*, 10(1) :23–38, 2010.

[115] Kaleigh Smith. Genetic Polymorphism and SNPs. Disponible sur `http://www.cs.mcgill.ca/~kaleigh/compbio/snp/snp_summary.html`, 2002.

[116] Ulrike Stege. *Resolving Conflicts in Problems from Computational Biology*. PhD thesis, ETH Zurich, Institute of Scientific Computing, 2000.

[117] Volker Turau and Bernd Hauck. A Self-Stabilizing Approximation Algorithm for Vertex Cover in Anonymous Networks. In *SSS '09 : Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 341–353. Springer-Verlag Berlin, Heidelberg, 2009.

[118] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[119] Fabien Vigier and Matthieu Latapy. Random Generation of Large Connected Simple Graphs with Prescribed Degree Distribution. In *COCOON '05 : Proceedings of the 11th International Conference on Computing and Combinatorics*, Kunming, Yunnan, Chine, 2005. Programmes disponibles sur `http://www-rp.lip6.fr/~latapy/FV/generation.html`.

[120] Sundar Vishwanathan. On Hard Instances of Approximate Vertex Cover. *ACM Transactions on Algorithms*, 5(1), 2008.

[121] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*, volume 2. Foundations and Trends in Theoretical Computer Science, Boston – Delft, 2009.

[122] Jeffrey Scott Vitter and Elisabeth A. M. Shriver. Algorithms for Parallel Memory I : Two Level Memories. *Algorithmica*, 12(2) :110–147, 1994.

[123] V. K. Wei. A Lower Bound on the Stability Number of a Simple Graph. Technical Report 81-11217-9, Bell Laboratories, 1981.

[124] Eric W. Weisstein. Butterfly graph. From MathWorld – A Wolfram Web Ressource : `http://mathworld.wolfram.com/ButterflyGraph.html`.

[125] Douglas B. West. *Introduction to Graph Theory, second edition.* Prentice Hall, 2001.

[126] Thomas K. F. Wong, Y. S. Chiu, T. W. Lam, and S. M. Yiu. Memory Efficient Algorithms for Structural Alignment of RNAs with Pseudoknots. *IEEE Transactions on Computational Biology and Bioinformatics*, 6(1), 2007.

[127] Andrew Chi-Chih Yao. Some Complexity Questions Related to Distributive Computing. In *STOC '79 : Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 209–213. ACM, 1979.

[128] Norbert Zeh. *Encyclopedia of Algorithms*, chapter I/O-Model. Springer-Verlag, 2008.

[129] Mariano Zelke. *Algorithms for Streaming Graphs.* PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät II, Humboldt-Universität zu Berlin, 2009.

[130] Mariano Zelke. Intractability of Min-Cut and Max-Cut in Streaming Graphs. *Information Processing Letters*, 111(3) :145–150, 2011.

[131] Daniel R. Zerbino and Ewan Birney. Velvet : Algorithms for De Novo Short Read Assembly Using de Bruijn Graphs. *Genome Research*, 18(5) :821–829, 2008.

[132] Yong Zhang, Qi Ge, Rudolf Fleisher, Tao Jiang, and Hong Zhu. Approximating the Minimum Weight Weak Vertex Cover. *Theoretical Computer Science*, 363(1) :99–105, 2006.

[133] Zhao Zhang, Xiaofeng Gao, and Weili Wu. PTAS for Connected Vertex Cover in Unit Disk Graphs. *Theoretical Computer Science*, 410(52) :5398–5402, 2009.