

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.

François Delbot

▶ **To cite this version:**

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.. Recherche opérationnelle [cs.RO]. Université d'Evry-Val d'Essonne, 2009. Français. NNT : . tel-00927315

HAL Id: tel-00927315
https://theses.hal.science/tel-00927315

Submitted on 12 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ D'EVRY
VAL D'ESSONNE

École doctorale Science et Ingénierie
U.F.R. Sciences Fondamentales et Appliquées — Université d'Évry Val d'Essonne

Thèse

présentée par

François Delbot

pour obtenir

le grade de docteur en sciences
de l'Université d'Évry Val d'Essonne
(spécialité informatique)

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.

---

soutenue publiquement le ***17 Novembre 2009*** *devant le jury composé de*

Rapporteurs :	M. Vangelis Paschos	(Professeur à l'Université Paris-Dauphine)
	M. Christophe Picouleau	(Professeur au CNAM)
Examinateurs :	M. Evripidis Bampis	(Professeur à l'Université d'Évry)
	M. Franck Butelle	(Maître de conférences à l'Université Paris Nord)
	M. Étienne Birmelé	(Maître de conférences à l'Université d'Évry)
Directeur de thèse :	M. Christian Laforest	(Professeur à l'Université Blaise Pascal)

Thèse préparée dans l'équipe OPAL au sein du laboratoire Informatique, Biologie Intégrative et Systèmes Complexes (IBISC) de l'Université d'Évry Val d'Essonne – FRE CNRS 3190

ibisc

Table des matières

Introduction générale **7**

I Comparaison et évaluation en moyenne d'algorithmes online et d'approximation pour le problème du vertex cover **11**

1 Le problème du vertex cover : les limites de l'évaluation en pire cas **13**
1.1 Le problème du vertex cover 14
1.2 Comparaison des deux algorithmes les plus connus 15
1.2.1 Les algorithmes Maximum Degree Greedy et Edge Deletion 15
1.2.2 Les limites de l'évaluation en pire cas 19
1.2.3 Les graphes Anti-MDG : un pire cas improbable 20
1.3 Bilan 23

2 Comparaison de deux algorithmes de liste **25**
2.1 Présentation des deux algorithmes de liste 26
2.2 L'algorithme Sorted ListRight est meilleur que l'algorithme Sorted ListLeft . 27
2.3 Les graphes *ProW* 31
2.4 Deux algorithmes facilement distribuables 32
2.4.1 Description des algorithmes SLL_{rep} et SLR_{rep} 34
2.4.2 Évaluation des deux algorithmes répartis 35
2.5 Bilan 37

3 Évaluation en moyenne de deux algorithmes de liste **39**
3.1 Sorted ListRight et Sorted ListLeft face au passé et au futur 39
3.2 LR est 2-approché en moyenne 43
3.3 Comparaison de ListRight et ListLeft sur les graphes d'Erdös-Renyi 46
3.3.1 Preuve du théorème 19 49
3.3.2 Espérance et variance de ListRight 52
3.4 Bilan 55

4 Calculs analytiques de l'espérance de la taille de vertex cover construits par divers algorithmes sur les chemins **57**
4.1 Deux algorithmes supplémentaires : Greedy Independent Cover et Depth First Search 58
4.2 Expression de l'espérance de ListLeft sur les chemins 60
4.3 Expression de l'espérance de ListRight sur les chemins 60
4.4 Expression de l'espérance de Edge Deletion sur les chemins 64

4.5 Expression de l'espérance de Maximum Degree Greedy sur les chemins 67
4.6 Répartition expérimentale des solutions retournées par les algorithmes étudiés 71
4.7 Bilan de l'évaluation du comportement des algorithmes sur les chemins 75

5 Comparaison expérimentale **83**
5.1 Que faut il mesurer ? 83
5.2 Démarche expérimentale 85
5.2.1 Méthode utilisée 85
5.2.2 Nos échantillons de tests 86
5.3 Les résultats 90
5.4 Bilan 98

6 Synthèse et perspectives de la première partie **99**

II Évaluation en moyenne du nombre de reconstructions d'un processus de mise-à-jour d'un arbre de connexion **103**

7 Analyse du nombre de perturbations lors du maintien d'un arbre de connexion de faible diamètre **105**
7.1 Introduction 105
7.1.1 Modélisation, contraintes et critère d'évaluation 107
7.1.2 État de l'art 109
7.2 Nombre d'étapes critiques au pire cas 110
7.3 Évaluation en moyenne du nombre de reconstructions d'un algorithme de mise-à-jour 111
7.3.1 Description du processus de mise-à-jour étudié. 111
7.3.2 Modélisation 112
7.3.3 Resultat principal : idée de preuve et interprétation 113
7.3.4 Preuve du théorème 40 114
7.4 Conclusion, perspectives 122

Conclusion générale **123**

Remerciements

Une thèse n'est pas une fin en soi, mais c'est un moment particulier dans la vie d'un chercheur. Je souhaite remercier tous ceux qui, de près ou de loin, ont contribué à ce travail car, si l'épreuve est individuelle, ses implications sont sociales, académiques, familiales, et humaines.

Je tiens à exprimer tout d'abord mes remerciements aux membres du jury, qui ont accepté d'évaluer mon travail de thèse.

Merci à M. Evripidis Bampis, Professeur à l'Université d'Évry, d'avoir accepté de présider le jury de cette thèse, à MM. les Professeurs Christophe Picouleau, Professeur des universités au CNAM et Vangelis Paschos, Professeur à l'Université Paris-Dauphine, d'avoir accepté d'être les rapporteurs de ce manuscrit. Leurs remarques et suggestions lors de la lecture de mon rapport m'ont permis d'apporter des améliorations à la qualité de ce dernier. Merci également à MM. Franck Butelle, maître de conférences à l'Université Paris Nord et Étienne Birmelé, maître de conférences à l'Université d'Évry, pour avoir accepté d'examiner mon mémoire et de faire partie de mon jury de thèse.

Je souhaite, évidemment, remercier mon directeur de thèse, M. Christian Laforest, car il a été le meilleur encadrant qu'un thésard puisse souhaiter avoir. Je me souviens que durant notre première rencontre, il m'avait expliqué que pour lui, un thésard est avant tout un collaborateur et non un subordonné. Ce n'est que plus tard que j'ai mesuré à quel point cette distinction est importante et avec quel sérieux il la respectait. Christian m'a guidé tout au long de ma thèse (dans une relation comme celles qu'entretiennent un Maître Jedi et son padawan) et il a contribué de manière significative à mon désir de poursuivre une carrière d'enseignant chercheur. Je souhaite le remercier plus que les mots ne peuvent le faire et j'espère pouvoir travailler avec lui le plus longtemps possible. Je remercie également Christine pour le réconfort que m'a apporté son délicieux saumon à un moment critique de la rédaction de mon mémoire.

Un grand, un très grand merci à Magalie, mon épouse, pour le soutien qu'elle m'a apporté au quotidien, sa patience (que l'on peut qualifier de surhumaine), ses petits plats, son organisation sans faille du cocktail qui a suivi ma soutenance et pour tant d'autres choses qui font de moi un homme comblé, tout simplement !

Merci à Stéphane, mon binôme depuis l'IUT (ça fait loin !), pour son aide et sa patience, de m'avoir servi de chauffeur et parfois de défouloir. Avoir réaliser toutes mes études supérieures avec toi a été un très grand plaisir et je suis persuadé que sans notre émulation, nous ne serions pas arrivés jusqu'ici !

Merci à mes parents et beaux parents, qui m'ont permis d'effectuer mes études sereinement. Je suis tout à fait conscient de ce que je leur doit. Merci à mes soeurs et à ma famille au sens large, vous savez bien pourquoi.

Enfin, parce que sans eux tout se serait certainement moins bien passé, je remercie chaleureusement les membres du laboratoire IBISC et les amis qui m'ont accompagné durant ces trois années de thèse. En particulier :

- les gens de la « salle café » pour les (longs, très longs, trop longs) repas animés, avec une pensée émue pour le moment où j'ai compris pourquoi l'an 0 n'a jamais existé tandis que l'an -1 si !
- les « anciens », Matthieu, Antoine, Stéfan, Assia, Delphine, Sylvie et François qui m'ont fait comprendre les rouages, les règles obscures qui régissent la vie d'un laboratoire...
- Laurent et Thomas, pour m'avoir fait découvrir que tard, le soir, les couloirs du laboratoire sont hantés et emplis de sons étranges et effrayants.
- ceux qui n'ont pas encore soutenu et à qui je souhaite bien du courage : Amandine, Ajitha, Romain, Fadi ;

Et enfin, Miss Tick, mon chat d'attaque qui m'aidera à dominer le monde, un jour, j'en suis sur !

A vous tous, MERCI !

Introduction générale

Un algorithme peut se définir comme l'énoncé d'une suite d'opérations permettant de donner la réponse à un problème précis. Ce concept, déjà utilisé par les Babyloniens puis par les géomètres grecs, a été systématisé par le mathématicien perse Abu Abdullah Muhammad ibn Musa al-Khwarizmi. Jusqu'à récemment (au regard de l'histoire), la majorité des algorithmes étaient utilisés pour trouver une solution exacte. L'objectif principal, lors de l'élaboration d'un tel algorithme, était donc de réduire au maximum le temps nécessaire au calcul de cette solution. Dans les années 60, on se contentait souvent d'exprimer le temps d'exécution en secondes d'un algorithme sur un jeu de données particulières. Cette façon de procéder rendait difficile la comparaison des algorithmes entre eux. Le temps d'exécution était dépendant du processeur utilisé, des temps d'accès à la mémoire vive, du langage de programmation, du compilateur utilisé, etc.

Une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. C'est l'objectif de la théorie de la complexité. Pour cela, on va lier le nombre d'opérations élémentaires effectuées par l'algorithme à une ou plusieurs variables du problème, le plus souvent il s'agira de la taille des données sur lesquelles on va appliquer l'algorithme. La théorie de la complexité algorithmique distingue les problèmes polynomiaux des problèmes NP-difficiles. Les problèmes polynomiaux sont des problèmes que l'on est capable de résoudre en un temps polynomial en la taille des données sur une machine de Türing déterministe, et on considère généralement qu'il s'agit d'un temps d'exécution « raisonnable ». Au contraire, on ne sait pas résoudre les problèmes NP-difficiles en un temps polynomial en la taille des données (et on ne le saura jamais si la conjecture $P \neq NP$ est vraie). En l'état actuel des connaissances, ces problèmes nécessitent un temps de résolution au moins exponentiel en la taille des données et on peut difficilement qualifier ce temps de raisonnable comme nous le montre cet exemple de Papadimitriou :

« Considérons un algorithme de complexité exponentielle, par exemple 2^n (n étant la taille des données). Pour $n = 100$ (une taille bien modeste pour un problème industriel), le nombre 2^{100} est plus grand que le nombre de milliardièmes de secondes qui se sont écoulés depuis le Big Bang jusqu'à aujourd'hui. »

Puisqu'il est improbable que l'on puisse trouver un algorithme polynomial résolvant de façon exacte un problème NP-difficile, l'utilisation d'algorithmes polynomiaux retournant des solutions non optimales s'est généralisée. Cependant, il est souhaitable que la qualité des solutions retournées ne soit pas trop dégradée et on demande donc des garanties sur leurs performances. La mesure de cette qualité se fait le plus souvent grâce au *rapport d'approximation en pire cas* et l'objectif devient donc de trouver, pour un problème donné, l'algorithme possédant le meilleur rapport d'approximation en pire cas possible. C'est ainsi que dans la littérature, on en vient à considérer qu'un algorithme est plus performant qu'un autre parce qu'il possède un meilleur rapport d'approximation. Cependant, il faut être conscient que cette mesure ne prend pas en compte la réalité de toutes

les exécutions possibles d'un algorithme. L'objectif de cette thèse, qui est découpée en deux parties, est d'évaluer certains algorithmes d'approximation de manière plus fine qu'avec le simple rapport d'approximation en pire cas.

Dans **la première partie**, nous allons considérer le problème du minimum vertex cover, un problème de minimisation NP-complet. Nous menons une étude comparative de différents algorithmes d'approximation et nous en profitons pour pointer du doigt certaines défaillances de l'approximation en pire cas. Le plan de cette partie est le suivant :

Le premier chapitre est consacré à une critique par l'exemple de l'évaluation en pire cas en utilisant deux des algorithmes d'approximation les plus connus (plus précisément, les plus couramment cités dans la littérature). Dans ce chapitre, nous montrons que cette mesure, bien qu'indispensable, n'est pas suffisante pour comparer les performances des algorithmes d'approximation et qu'il est nécessaire de recourir à des méthodes de comparaison plus fines.

Le deuxième chapitre est consacré à la présentation d'un nouvel algorithme de liste triée* et à sa comparaison avec celui proposé récemment par Avis et Imamura dans [AI07]. Nous montrons que pour toute instance, notre algorithme retourne une meilleur solution.

Par ailleurs, nous avons proposé une version répartie des deux algorithmes et nous les comparons sur d'autres critères (nombre de messages échangés et nombre d'étapes de communication).

Le troisième chapitre étudie l'effet sur ces deux mêmes algorithmes du relâchement de la contrainte de tri. Nous évaluons leurs performances en moyenne, sous des hypothèses d'équiprobabilité et nous donnons une borne supérieure de l'espérance de la taille des solutions retournées par notre algorithme. Cette étude est suivie de l'évaluation des performances moyennes de ces deux algorithmes sur les graphes aléatoires de Erdös-Renyi.

Le quatrième chapitre contient une étude analytique du comportement moyen exact de différents algorithmes sur une classe de graphe simple mais non triviale : les chemins. Pour plusieurs algorithmes, nous mènerons ensuite une étude expérimentale de la répartition des tailles de solutions obtenue après un grand nombre d'itérations d'un même algorithme.

Le cinquième chapitre est une étude comparative expérimentale des différents algorithmes déjà observés depuis le début de la thèse. Dans un premier temps, nous définissons la mesure qui nous semble la plus adéquate pour capturer la qualité expérimentale d'un algorithme ainsi que les différents échantillons de graphes sur lesquels nous mènerons les expérimentations. La suite du chapitre est consacrée à l'exploitation des résultats.

Tous ces travaux, analytiques et expérimentaux, montrent que juger de la qualité d'un algorithme uniquement en fonction de son rapport d'approximation en pire cas est une erreur. En effet, les algorithmes ayant le meilleur rapport d'approximation sont ceux qui ont le plus mauvais comportement en moyenne. Il convient donc de compléter l'évaluation en pire cas par une étude en moyenne des performances des algorithmes.

La deuxième partie de cette thèse poursuit des travaux menés précédemment par Nicolas Thibault et Christian Laforest. L'objectif de cette partie est d'aller plus loin qu'un résultat d'impossibilité en pire cas en menant une évaluation en moyenne.

*il s'agit d'un algorithme d'approximation basé sur un ordre statique des sommets déterminé par leurs degrés.

Plus précisément, nous nous intéressons dans cette partie aux communications entre les membres d'un groupe de machines que l'on va interconnecter par un arbre, bâti sur le réseau sous-jacent. Une caractéristique importante d'un groupe au sein d'un réseau est qu'il est en constante évolution. On peut en effet évoquer toute forme de forum ou de réunion via un réseau. Dans ce type de rassemblements virtuels, les participants peuvent arriver et/ou partir, dans un ordre et à des moments inconnus à l'avance. Par exemple, dans les systèmes pair à pair, il est inconcevable de prédire qui va échanger des données avec qui. Les échanges se font « au fil de l'eau », en fonction de paramètres qu'il est difficile de quantifier à l'avance. Par conséquent, connaître à l'avance la totalité des membres du groupe à couvrir/connecter n'est pas toujours possible en pratique. Nous nous sommes donc focalisés sur le cas dynamique qui consiste à incrémenter et/ou décrémenter un arbre de connexion pour prendre en compte les arrivées et départs de membres. Cependant, il est naturel d'exiger que cet arbre maintienne, à chaque étape, un niveau de qualité satisfaisant. Plus précisément, nous fixons la contrainte suivante : à chaque requête (ajout ou retrait), le diamètre de l'arbre courant doit être au plus deux fois le diamètre minimum. Pour garantir cela, nous devons permettre la reconstruction totale de l'arbre à certaines étapes. Une telle reconstruction nécessite, d'un point de vue réseau, la mise à jour des tables de routage de chaque membre du groupe, perturbant ainsi les communications en cours et induisant un fort trafic de contrôle. L'objectif est alors de minimiser le nombre d'étapes où ont lieu de telles reconstructions.

Il a été montré (voir la thèse de Nicolas Thibault [Thi06]) que dans le pire des cas, le nombre de reconstructions nécessaires est linéaire en le nombre d'événements (ajouts et/ou retraits). Ce résultat en pire cas est intrinsèquement lié aux contraintes que nous imposons et est valable pour *n'importe quel algorithme* de mise-à-jour les respectant. Il s'agit donc ici d'une limitation fondamentale. Cependant, cette limitation est assez artificielle dans la mesure où le système servant à la décrire et la suite des événements qui conduisent à cette situation sont très spécifiques et *a priori* rares. Cette rareté apparente nous a conduits à nous poser le problème de l'évaluation de ce paramètre *en moyenne*. Ainsi, après avoir proposé un procédé simple de mise-à-jour, nous démontrons (en utilisant des marches aléatoires) qu'il induit en moyenne un nombre au plus proportionnel en le *logarithme* du nombre d'événements ce qui fait de lui un procédé intéressant à déployer.

Remarque sur la lecture du mémoire. Les deux parties de la thèse sont totalement indépendantes et peuvent donc être lues dans n'importe quel ordre.

Première partie

Comparaison et évaluation en moyenne d'algorithmes online et d'approximation pour le problème du vertex cover

Chapitre 1

Le problème du vertex cover : les limites de l'évaluation en pire cas

Les algorithmes d'approximation sont, le plus souvent, associés aux problèmes NP-difficiles. En effet, puisqu'il est improbable que l'on puisse trouver un algorithme polynomial résolvant de façon exacte un problème NP-difficile, l'utilisation d'algorithmes polynomiaux retournant des solutions non optimales s'est généralisée. Cependant, il est souhaitable que la qualité des solutions retournées ne soit pas trop dégradée et on demande donc à ces algorithmes de donner une garantie sur leurs performances.

Par exemple, pour un problème de minimisation, un algorithme $\alpha-$approché est un algorithme qui s'exécute (généralement) en un temps qui est polynomial en la taille de l'instance à traiter et qui retourne une solution dont on garantit que la valeur ne dépasse pas α fois la valeur de la solution optimale. Autrement dit, si on note Opt la valeur de la solution optimale, et Sol la valeur de la solution retournée par l'algorithme, on a $Opt \leq Sol \leq \alpha Opt$.

A partir de cette définition, l'objectif devient donc de trouver, pour un problème donné, l'algorithme possédant le meilleur rapport d'approximation en pire cas possible. C'est ainsi que dans la littérature, on en vient à considérer qu'un algorithme est plus performant qu'un autre parce qu'il possède un meilleur rapport d'approximation. Cependant, il faut être conscient que cette mesure ne prend pas en compte la réalité de toutes les exécutions possibles d'un algorithme. On peut imaginer, par exemple et de manière extrêmement caricaturale, qu'un algorithme soit optimal pour toutes les instances possibles sauf une pour laquelle il retournera une solution dont le rapport d'approximation sera très grand (plus grand que celui des autres algorithmes d'approximation pour le même problème). Du point de vue de l'évaluation en pire cas, cet algorithme sera très mauvais alors que ses performances globales sont excellentes. Un autre problème de l'évaluation en pire cas provient du fait que pour une instance particulière, un rapport d'approximation en pire cas de α (pour un problème de minimisation par exemple) peut situer la valeur d'une solution approchée au delà de la valeur de la pire solution possible pour cette instance.

Dans ce chapitre, nous allons illustrer (concrètement) certaines défaillance de l'évaluation en pire cas en utilisant le problème du vertex cover ainsi que deux de ses algorithmes d'approximation les plus connus (plus précisément, les plus couramment cités dans la littérature). Par ces résultats, nous verrons que l'évaluation en pire cas, bien qu'indispensable, n'est pas suffisante pour comparer les performances des algorithmes d'approximation, justifiant ainsi les travaux de cette thèse, comme

la mise en lumière de relation de dominance entre deux algorithmes (l'un des deux est toujours meilleur que l'autre) ou l'évaluation en moyenne de la tailles des solutions retournées.

1.1 Le problème du vertex cover

En 1972, un an après que Cook [Coo71] ait formalisé la notion de NP-complétude et prouvé que le problème SAT est NP-complet, Richard Karp [Kar72] a publié un article fondateur en théorie de la complexité, dans lequel il prouve la NP-complétude de 21 problèmes dont celui du Minimum Vertex Cover (MVC) :

Étant donné un graphe $G = (V, E)$, avec V l'ensemble de ses sommets et E l'ensemble de ses arêtes, le problème du vertex cover consiste à trouver le plus petit ensemble possible $C \subseteq V$ tel que pour toute arête $(i, j) \in E$ on ait $i \in C$ ou $j \in C$ (ou bien les deux). On dit que C est une couverture minimale de G (un vertex cover de G).

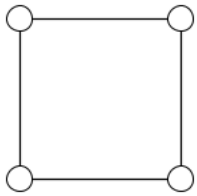


FIG. 1.1 – Un graphe G

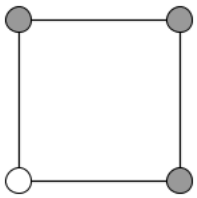


FIG. 1.2 – Une couverture des sommets de G

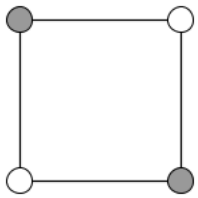


FIG. 1.3 – Une couverture optimale des sommets de G

Par exemple, la figure 1.1 représente un graphe avec 4 sommets et 4 arêtes. Les sommets en gris des figures 1.2 et 1.3 sont les sommets faisant partie de la solution. Clairement, les sommets de la figure 1.2 forment bien une couverture (un vertex cover) car toutes les arêtes possèdent au moins une extrémité dans l'ensemble des sommets gris, mais elle n'est pas de taille minimum et n'est pas non plus minimale*. La figure 1.3 propose un vertex cover minimum (et cette solution est donc minimale).

Le fait que ce problème soit NP-complet signifie qu'il est très improbable qu'on puisse trouver un algorithme le résolvant de manière exacte en un temps polynomial. De plus, ce problème reste NP-complet même pour certaines classes de graphes particulières (par exemple les graphes de degré borné [GJS74]).

La recherche d'un vertex cover intervient souvent dans la résolution d'autres problèmes (le pus souvent lors d'une étape intermédiaire) comme par exemple dans l'alignement de séquences multiples [RK00], la résolution de conflits biologiques [Ste00] ou bien encore dans la surveillance des réseaux [ZGF+06]). Ces applications nécessitent des méthodes de résolution différentes (résolution exacte, heuristique ou d'approximation avec garantie de performances) le choix s'effectuant en fonction du temps dont on dispose et de la qualité de solution requise. Ainsi, de nombreuses heuristiques

*On distingue le terme minimum qui représente ici la valeur de la plus petite solution du terme minimal que l'on utilisera au sens de l'inclusion. Dans le cas présent, cette solution n'est pas minimale car si on retire le sommet gris en haut et à droite du graphe, notre solution est toujours une couverture. Évidemment, une solution de taille minimum est forcément minimale.

ont été proposées [KB94, YK98, KG03, XCW04, SJSL04, XM06] et bien que ces algorithmes ne garantissent pas de trouver la meilleure solution, leur temps d'exécution est acceptable.

Le problème du MVC appartient à la classe FPT (Fixed Parameter Tractable), ce qui signifie qu'on peut le résoudre de manière exacte en un temps qui est exponentiel, non pas en la taille de l'instance mais en k, la taille de la couverture optimale. Cette complexité paramétrique a suscité beaucoup d'intérêt [BFR98, CKJ01, DF98, DFS97, NR00, BFR98] et plusieurs algorithmes exacts ont ainsi été proposés [DF95, CKJ01, CDRC$^+$03]. Ces algorithmes exacts sont tout à fait efficaces étant donné que pour de nombreuses applications le paramètre k est petit [AGN01, DF98, DFS97, Fel01] et le meilleur algorithme connu s'exécute en $O(1.2852^k + kn)$ [CKJ01]. Cependant, ces algorithmes ne sont pas utilisables en pratique dès que le paramètre k devient trop grand et il convient donc d'utiliser les algorithmes d'approximation.

Le problème du vertex cover appartient à la classe APX, ce qui signifie qu'il admet un algorithme polynomial dont le rapport d'approximation est borné par une constante. L'un de ces algorithmes d'approximation, basé sur les couplages maximaux, donne un rapport d'approximation de 2 [GJ79]. Malgré de nombreux efforts, très peu de progrès ont été fait pour améliorer ce rapport d'approximation. Jusqu'à aujourd'hui, aucun algorithme dont le rapport d'approximation soit borné par une constante inférieure à 2 n'a été trouvé. Monien et Speckenmeyer [MS85] et Bar-Yehuda et Even [BYE85] ont proposé des algorithmes avec un rapport d'approximation de $2 - \frac{\ln \ln n}{\ln n}$, avec n le nombre de sommets du graphe et Karakostas [Kar05] a réduit ce rapport d'approximation à $2 - \Theta(\frac{1}{\sqrt{logn}})$.

1.2 Comparaison des deux algorithmes les plus connus

1.2.1 Les algorithmes Maximum Degree Greedy et Edge Deletion

Dans cette partie, nous présentons les deux algorithmes d'approximation les plus connus pour le problème du vertex cover. Le premier possède un rapport d'approximation en pire cas de 2 que l'on peut qualifier d'optimal si la conjecture de [KR08] est vérifié. Le second possède un « mauvais » rapport d'approximation en pire cas, mais l'heuristique sur laquelle il se fonde est sûrement la première à laquelle on pense lorsque l'on souhaite résoudre le problème du vertex cover.

L'algorithme Edge Deletion (ED), proposé par Gavril (voir [GJ79]), est basé sur le fait que les sommets d'un couplage maximal (au sens de l'inclusion, c'est-à-dire une ensemble d'arêtes deux à deux non adjacentes tel qu'on ne puisse plus ajouter d'autre arête du graphe) forment une couverture des sommets.

Algorithme 1 : Edge Deletion

Données : Un graphe $G = (V, E)$
Sortie : Un vertex cover de G
$C \leftarrow \emptyset$;
tant que $E \neq \emptyset$ **faire**
  sélectionner $uv \in E$;
  $C \leftarrow C \cup \{u, v\}$;
  $V \leftarrow V - \{u, v\}$;
fin tant que
retourner C;

L'algorithme Edge Deletion retourne un vertex cover dont la taille est au plus $2 \cdot |OPT|$ (voir [CLRS90]) et s'exécute en un temps linéaire au nombre d'arêtes du graphe. Récemment, il a été montré que son rapport d'approximation en pire cas est asymptotique à $\min\{2, \frac{1}{1-\sqrt{1-\epsilon}}\}$ pour les graphes ayant un degré moyen d'au moins ϵn et à $\min\{2, \frac{1}{\epsilon}\}$ pour les graphes ayant un degré minimum d'au moins ϵn. (voir [CLL$^+$05])

La figure 1.4 présente un graphe pour lequel Edge Deletion retourne toujours une solution (représentée par les arêtes doubles) qui est 2−approchée, tandis que la figure 1.5 présente un graphe pour lequel Edge Deletion *peut* retourner une solution optimale (représentée elle aussi par une arête double).

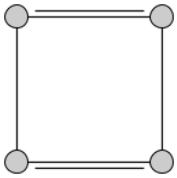


Fig. 1.4 – Exemple de graphe pour lequel Edge Deletion atteint son rapport d'approximation en pire cas

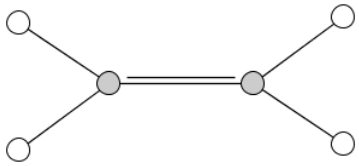


Fig. 1.5 – Exemple de graphe pour lequel Edge Deletion peut retourner une solution optimale

Dans la boucle de l'algorithme Edge Deletion, le choix de l'arête à ajouter au couplage peut s'effectuer de plusieurs façons. On peut sélectionner une arête parmi l'ensemble des arêtes disponibles (c'est le choix qu'effectue l'algorithme que nous utilisons), mais il est possible de sélectionner un sommet non isolé, puis de sélectionner un de ses voisins. Cette dernière méthode semble retourner des couplages de taille plus importante que la première (voir [DFP93]) et donc des tailles de vertex cover plus importantes, c'est pour cette raison que nous ne l'utilisons pas ici.

Étant donné qu'on ne connaît pas d'algorithme dont le rapport d'approximation en pire cas est borné par une constante meilleure que 2, et si la conjecture selon laquelle il n'en existe pas est vérifiée (voir [KR08]), alors on peut dire que Edge Deletion a atteint une certaine forme d'optimalité. Dans la littérature, cet algorithme est considéré comme le meilleur algorithme connu (voir [Pas97] par exemple).

L'algorithme Maximum Degree Greedy (MDG) est sûrement le premier auquel on pense lorsqu'on essaye de résoudre le problème de la couverture. En effet, l'algorithme va couvrir le maximum d'arêtes possibles à chaque étape en sélectionnant le sommet de degré maximum :

Algorithme 2 : Maximum Degree Greedy

Données : Un graphe $G = (V, E)$
Sortie : Un vertex cover de G
$C \leftarrow \emptyset$;
tant que $E \neq \emptyset$ **faire**
  sélectionner un sommet u de degré maximum;
  $V \leftarrow V - \{u\}$;
  $C \leftarrow C \cup \{u\}$;
fin tant que
retourner C;

L'algorithme Maximum Degree Greedy retourne un vertex cover dont la taille est au plus $H(\Delta) \cdot |OPT|$, avec $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ la série harmonique et Δ le degré maximum du graphe (par adaptation de la preuve présente dans [CLRS90] pour le problème de la couverture d'ensemble).

L'heuristique de cet algorithme est basée sur la considération suivante :

Soit G un graphe possédant un sommet de degré supérieur à k, avec k la taille d'une couverture minimum de G. Supposons qu'un algorithme choisisse de ne pas sélectionner le sommet de degré maximum. Il va donc être obligé de sélectionner tout son voisinage, sinon certaines arêtes ne seraient pas couvertes. Or, la taille du voisinage étant supérieure à k, l'algorithme est sûr de ne pas retourner une solution optimale. Outre le fait que Maximum Degree Greedy maximise le nombre d'arêtes couvertes à chaque étape, sélectionner tous les sommets dont le degré est supérieur à la taille de la couverture minimum semble être une bonne idée. Les figures 1.6 et 1.7 nous donnent deux exemples de graphes pour lesquels Maximum Degree Greedy retourne toujours une solution optimale.

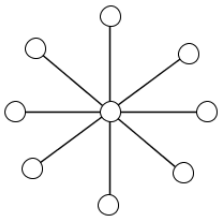


Fig. 1.6 – Étoile

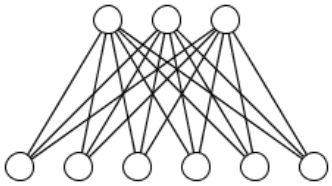


Fig. 1.7 – Graphe biparti complet

Les graphes Anti-MDG sont conçus pour piéger l'algorithme Maximum Degree Greedy et atteindre son rapport d'approximation en ordre de grandeur. Nous allons voir comment construire ces graphes (en nous appuyant sur un exemple), ce qui nous permettra d'effectuer l'analyse du comportement de Maximum Degree Greedy sur ces graphes.

On note k un entier qui représente la dimension du graphe Anti-MDG (dans notre exemple, on a $k = 6$). Ce graphe, que l'on nommera G, possède trois ensembles de sommets A, B et C. On note X_i le $i$ème sommet de l'ensemble X, avec $X \in \{A, B, C\}$.

Au départ, les ensembles A et B contiennent exactement k sommets chacun et l'ensemble C ne contient aucun sommet. On crée un couplage entre les sommets de A et ceux de B : chaque sommet B_i est relié au sommet A_i, pour $i = 1, \ldots, k$ (voir figure 1.8).

Nous allons ensuite ajouter $\lfloor \frac{k}{2} \rfloor$ sommets de degré 2 dans C. Le premier sera relié aux deux premiers sommets de B, le deuxième aux deux suivants et ainsi de suite. Plus formellement, nous allons relier les sommets B_{2i+1} et B_{2i+2} au sommet C_{i+1}, pour i allant de 0 à $\lfloor \frac{k}{2} \rfloor - 1$ (voir figure 1.9).

Ensuite, nous ajoutons $\lfloor \frac{k}{3} \rfloor$ sommets de degré 3 dans C. Le premier sera relié aux trois premiers sommets de B, le deuxième aux trois suivants et ainsi de suite (voir figure 1.10). Plus formellement, on relie les sommets B_{3i+1}, B_{3i+2} et B_{3i+3} au sommet C_{n+i+1}, pour i allant de 0 à $\lfloor \frac{k}{3} \rfloor - 1$, avec $n = \lfloor \frac{k}{2} \rfloor$ le nombre de sommets présents dans C avant l'ajout des sommets de degré 3. On recommence une nouvelle fois l'opération mais en ajoutant cette fois-ci $\lfloor \frac{k}{4} \rfloor$ sommets de degré 4 dans C (voir figure 1.11). Notons que dans notre exemple, nous n'ajoutons qu'un seul sommet.

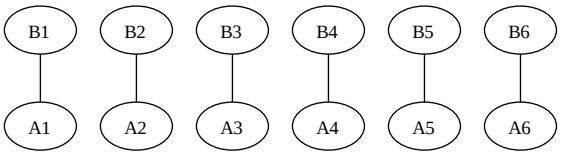


FIG. 1.8 – Première étape : création d'un couplage entre les sommets de A et B.

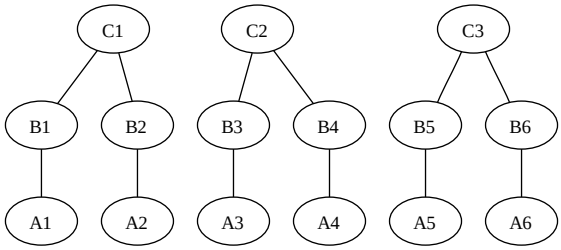


FIG. 1.9 – Deuxième étape : on ajoute les sommets C_1, C_2 et C_3 qui sont de degré 2.

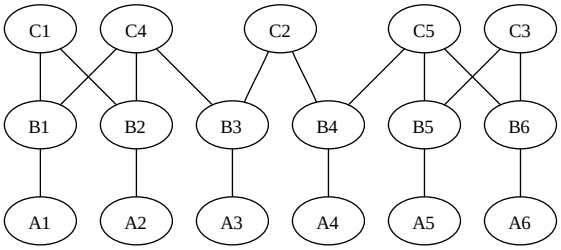


FIG. 1.10 – Troisième étape : on ajoute les sommets C_4 et C_5 qui sont de degré 3.

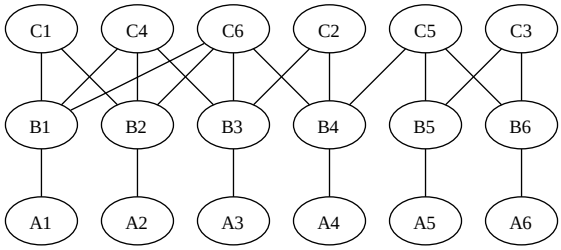


FIG. 1.11 – Quatrième étape : on ajoute le sommet C_6 de degré 4

On poursuit de la même façon jusqu'à ce qu'on relie les k sommets de B à un unique sommet de C. Dans notre exemple, nous pouvons donc encore ajouter un sommet de degré 5 (voir figure 1.12) et un sommet de degré 6.

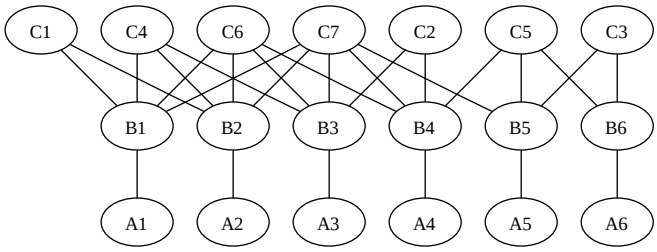


FIG. 1.12 – Cinquième étape : on ajoute le sommet C_7 de degré 5

Le graphe final G (voir figure 1.13) est biparti, son degré maximum est k et pour tout sommet $u \in A$, on a $d(u) = 1$.

Dans un tel graphe, il est clair que l'ensemble B est un vertex cover de taille k. L'algorithme ED peut retourner tous les sommets de B et de A (puisque l'ensemble des arêtes (B_i, A_i), pour $i = 1, \ldots, k$ forme un couplage). ED peut donc retourner une solution de taille $2k$. Comme ED est un algorithme $2-$approché, on obtient que $|OPT| = k$.

Comportement de Maximum Degree Greedy sur les graphes Anti-MDG. Il existe une exécution de l'algorithme MAXIMUM DEGREE GREEDY ([PY88]) qui, dans un premier temps, sélectionne tous les sommets de l'ensemble C, puis sélectionne k sommets parmi ceux de l'ensemble $B \cup A$ (pour couvrir les arêtes (B_i, A_i)). Pour s'en convaincre, il suffit de remarquer que lors de

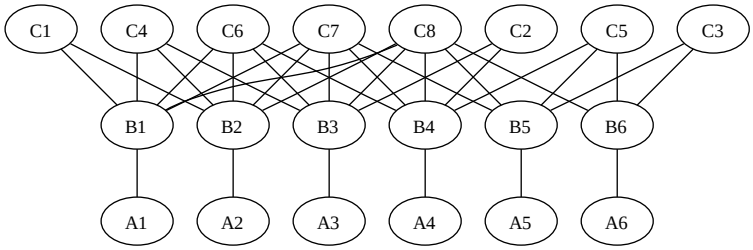


FIG. 1.13 – Après l'ajout du sommet C_8 de degré 6, on obtient le graphe Anti-MDG de dimension 6

la construction du graphe, de l'étape 2 jusqu'à la dernière étape, il existe un sommet de degré maximum dans l'ensemble C. MAXIMUM DEGREE GREEDY peut donc sélectionner le sommet de degré k de l'ensemble C, puis le sommet de degré $k-1$ et ainsi de suite, repassant par chacune des étapes de construction dans l'ordre inverse. Pour les étapes où nous avons ajouté plusieurs sommets, l'algorithme peut les sélectionner à la suite et dans un ordre quelconque, car les sommets de C forment un ensemble indépendant et la suppression d'un sommet ne diminuera pas le degré des autres sommets de C. Lorsqu'il n'y a plus de sommet dans l'ensemble C, MAXIMUM DEGREE GREEDY va devoir couvrir chaque arête du couplage de la première étape et va donc retourner une solution de taille $\left(k + \left\lfloor \frac{k}{2} \right\rfloor + \left\lfloor \frac{k}{3} \right\rfloor + \left\lfloor \frac{k}{4} \right\rfloor + \ldots + 1\right) \approx H(k) \cdot k = H(\Delta) \cdot |OPT|$, avec Δ le degré maximum du graphe et H la série harmonique.

1.2.2 Les limites de l'évaluation en pire cas

Nous venons de voir deux algorithmes parmi les plus connus pour le problème de la couverture de sommets. Ces deux algorithmes offrent des garanties de performances très différentes : EDGE DELETION garantit qu'il ne retournera jamais une solution dont la taille est supérieure à deux fois la taille de la solution optimale, tandis que MAXIMUM DEGREE GREEDY garantit une $H(\Delta)$ approximation, ce qui est relativement mauvais surtout lorsque le degré du graphe est grand. Évidemment, ce n'est pas parce qu'un algorithme possède un meilleur rapport d'approximation qu'il est toujours meilleur. Par définition, un rapport d'approximation en pire cas de k garantit seulement que la taille de la solution retournée se situe dans l'intervalle $[|OPT|, k|OPT|]$.

Si nous considérons la figure 1.6, MAXIMUM DEGREE GREEDY va toujours retourner une solution optimale alors que son rapport d'approximation est de $H(8) \approx 2.71$ et EDGE DELETION va toujours retourner deux sommets, soit un rapport d'approximation de 2. EDGE DELETION possède un meilleur rapport d'approximation que MAXIMUM DEGREE GREEDY, mais il retourne toujours une solution deux fois plus mauvaise que MAXIMUM DEGREE GREEDY.

La grande majorité des algorithmes pour le vertex cover possèdent une part d'indéterminisme. Par exemple, MAXIMUM DEGREE GREEDY doit sélectionner un sommet de degré maximum, et lorsque plusieurs sommets sont de degré maximum, il n'est pas précisé comment choisir le sommet à sélectionner. De la même façon, EDGE DELETION choisit les sommets d'une arête et nous avons toute latitude pour choisir l'arête. L'évaluation en pire cas force les algorithmes à effectuer le plus mauvais choix possible, ce qui ne reflète pas toujours le comportement global de l'algorithme. Ce constat nous amène à vouloir évaluer les performances des algorithmes de manière plus fine, en prenant en compte leur part d'indéterminisme pour coller au plus juste, au plus proche de leur comportement concret.

Évaluer l'influence de l'indéterminisme d'un algorithme sur la qualité des solutions qu'il retourne peut amener à des résultats surprenants, comme nous allons le voir avec MAXIMUM DEGREE GREEDY.

1.2.3 Les graphes Anti-MDG : un pire cas improbable

Dans cette partie, nous montrons que MAXIMUM DEGREE GREEDY possède un très bon comportement moyen sur les graphes qui ont été spécialement conçus pour le piéger en pire cas. Plus précisément, le lemme 5 nous indique que l'espérance du rapport d'approximation des solutions qu'il retourne tend vers 1 lorsque la dimension du graphe Anti-MDG tend vers l'infini. Pour montrer cela, nous avons besoin des lemmes suivants :

Lemme 1 *Dans un graphe Anti-MDG de dimension k, les sommets B_i, $i \in \{1 \dots \lfloor \frac{k}{2} \rfloor\}$, sont de degré k.*

Preuve. Comme nous l'avons vu dans la partie consacrée à la construction des graphes Anti-MDG, la i-ème étape concerne l'ajout de $\lfloor \frac{k}{i} \rfloor$ sommets de degré i dans l'ensemble C. Le premier sommet créé de cette manière est relié aux i premiers sommets de B, le deuxième aux i suivants et ainsi de suite. Donc, par construction, les $i \cdot \lfloor \frac{k}{i} \rfloor$ premiers sommets de B sont reliés à un sommet de C de degré i, avec $i \in \{1 \dots k\}$. Cette remarque est aussi valable pour la première étape car chaque sommet de B est relié à un sommet différent de A.

Montrons que $i \cdot \lfloor \frac{k}{i} \rfloor \geq \lfloor \frac{k}{2} \rfloor$, avec $i \in \{1 \dots k\}$:

On pose $k = q \cdot i + r$, avec $0 \leq r < i$ le reste de la division entière de k par i. On a $i > r$ et comme $i \in \{1 \dots k\}$ on obtient que $q \geq 1$ et donc que $i \cdot q > r$.

$$\Rightarrow 2 \cdot i \cdot q > i \cdot q + r \Rightarrow i \cdot q > \frac{i \cdot q + r}{2} \geq \left\lfloor \frac{i \cdot q + r}{2} \right\rfloor = \left\lfloor \frac{k}{2} \right\rfloor$$

$$\text{et comme } i \cdot \left\lfloor \frac{k}{i} \right\rfloor = i \cdot \left\lfloor \frac{i \cdot q + r}{i} \right\rfloor = i \cdot q, \text{ on obtient que } i \cdot \left\lfloor \frac{k}{i} \right\rfloor \geq \left\lfloor \frac{k}{2} \right\rfloor$$

Comme $i \cdot \lfloor \frac{k}{i} \rfloor \geq \lfloor \frac{k}{2} \rfloor$, cela signifie que lors de chaque étape d'ajout de sommets, nous allons augmenter le degré des $\lfloor \frac{k}{2} \rfloor$ premiers sommets de B de 1. Par construction, le nombre de ces étapes est de k, ce qui démontre que les $\lfloor \frac{k}{2} \rfloor$ premiers sommets de B sont de degré k. ■

Lemme 2 *Soit G' le graphe résultant de la suppression des $s \geq 1$ premiers sommets de l'ensemble B d'un graphe Anti-MDG et v_Δ, le sommet de degré maximum de G'. On a $v_\Delta \in B$ et $v_\Delta \notin C$.*

Preuve. Soit G un graphe Anti-MDG de dimension k, et G' le graphe résultant de la suppression dans G des s premiers sommets de B. On note A', B' et C' les ensembles de sommets correspondants dans G'. On a $|B'| = k - s$, et comme les sommets de l'ensemble C' ne sont reliés qu'à des sommets de B', on obtient que le degré maximum d'un sommet dans C' est de $k - s$.

Nous allons montrer que le premier sommet de B' est de degré au moins $k - s + 1$. Notons C_{si} le sommet de C de degré $s + i$ dans le graphe initial. Par construction, dans G', le sommet C_{si}, avec $i \in \{1 \dots k - s\}$, est relié aux i premiers sommets de B' présents dans G'. Le premier sommet de B' est donc relié aux sommets C_{si},avec $i \in \{1 \dots k - s\}$. Ce sommet étant aussi relié à un sommet de A', il est donc de degré au moins $k - s + 1$. Le degré maximum d'un sommet de C' étant $k - s$, le sommet de degré maximum se trouve forcément dans B'. ■

Lemme 3 *Soit u le premier sommet sélectionné par* Maximum Degree Greedy *sur un graphe Anti-MDG. Si $u \in B$, alors l'algorithme va sélectionner l'intégralité des sommets de B, et seulement eux.*

Preuve. Le lemme 1 nous indique que dans un graphe Anti-MDG de dimension k, les $\lfloor \frac{k}{2} \rfloor$ premiers sommets de B sont de degré k. Comme il s'agit du degré maximum du graphe, Maximum Degree Greedy peut donc sélectionner l'un de ces sommets en premier. Le lemme 2 montre que si un sommet de B est retiré du graphe, alors le sommet de degré maximum se trouve forcement dans B (et pas dans C), ce qui impose à Maximum Degree Greedy de sélectionner tous les sommets de B (par applications successives du lemme 2). Lorsque tous les sommets de B ont été sélectionnés, il ne reste plus d'arêtes dans le graphe et l'algorithme termine. ■

Lemme 4 *Une majoration de l'espérance de la taille des solutions retournées par* Maximum Degree Greedy *sur un graphe Anti-MDG de dimension k est :*

$$\frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} \cdot k + \frac{1}{\lfloor \frac{k}{2} \rfloor + 1} \cdot \sum_{i=1}^{k} \left\lfloor \frac{k}{i} \right\rfloor$$

Preuve. Le lemme 1 nous indique qu'il y a au moins $\lfloor \frac{k}{2} \rfloor$ sommets de degré k dans B tandis que par construction, il n'en existe qu'un seul dans C.

Si Maximum Degree Greedy sélectionne un sommet de B en premier, alors il va retourner les sommets de B (lemme 3), soit k sommets et on suppose que si l'algorithme sélectionne le sommet de C lors de la première étape alors il va retourner la pire solution possible (ce qui est une majoration grossière).

En faisant l'hypothèse que lorsque plusieurs sommets sont de degré maximum, Maximum Degree Greedy va effectuer son choix de manière équiprobable, la probabilité qu'un sommet de B soit sélectionné lors de la première étape est $\frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1}$ et la probabilité de sélectionner le sommet de C est $\frac{1}{\lfloor \frac{k}{2} \rfloor + 1}$.

On en déduit une majoration de l'espérance de la taille d'une couverture retournée par Maximum Degree Greedy :

$$\frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} \cdot k + \frac{1}{\lfloor \frac{k}{2} \rfloor + 1} \cdot \sum_{i=1}^{k} \left\lfloor \frac{k}{i} \right\rfloor$$

■

Lemme 5 *L'espérance du rapport d'approximation de* Maximum Degree Greedy *appliqué sur un graphe Anti-MDG de dimension k tend vers 1 lorsque k tend vers $+\infty$.*

Preuve. Dans un graphe Anti-MDG de dimension k, la taille d'une couverture optimale est de k (voir la description de leur construction). En reprenant le raisonnement de la preuve du lemme 4, on obtient que Maximum Degree Greedy va retourner une solution de taille k avec probabilité $\frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1}$ et une solution de taille $\sum_{i=1}^{k} \lfloor \frac{k}{i} \rfloor$ avec probabilité $\frac{1}{\lfloor \frac{k}{2} \rfloor + 1}$. On en déduit une majoration de l'espérance de son rapport d'approximation :

$$\frac{k}{k} \cdot \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{\sum_{i=1}^{k} \lfloor \frac{k}{i} \rfloor}{k} \cdot \frac{1}{\lfloor \frac{k}{2} \rfloor + 1}$$

$$\leq \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{1}{k} \frac{1}{\lfloor \frac{k}{2} \rfloor + 1} \sum_{i=1}^{k} \frac{k}{i}$$

$$= \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{1}{\lfloor \frac{k}{2} \rfloor + 1} \cdot \sum_{i=1}^{k} \frac{1}{i}$$

$$\leq \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{1}{\frac{k}{2}} \cdot \sum_{i=1}^{k} \frac{1}{i}$$

Et comme $H(n) \approx \ln(n)$ quand n est grand, on obtient :

$$\lim_{k \to \infty} \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{2}{k} \cdot \sum_{i=1}^{k} \frac{1}{i} = \lim_{k \to \infty} \frac{\lfloor \frac{k}{2} \rfloor}{\lfloor \frac{k}{2} \rfloor + 1} + \frac{2 \ln(k)}{k} = 1$$

■

Ce dernier résultat nous montre que même si Maximum Degree Greedy possède une exécution catastrophique en pire cas sur les graphes Anti-MDG, son comportement moyen est très bon : il retourne une solution optimale avec très forte probabilité et son rapport d'approximation est très proche de 1. Ces performances sont d'autant plus étonnantes que cette classe de graphes a été spécialement conçue pour piéger l'algorithme et l'ordre de grandeur du rapport d'approximation en pire cas de Maximum Degree Greedy est logarithmique en la dimension du graphe (son rapport d'approximation en pire cas tend donc vers l'infini). Ce comportement contre intuitif illustre parfaitement une des lacunes de l'évaluation en pire cas.

On peut remarquer trivialement que toute exécution de Edge Deletion retourne exactement $2k$ sommets, ce qui est bien plus mauvais que Maximum Degree Greedy, en moyenne du moins.

Il est néanmoins possible de définir une classe de graphes telle que toute exécution de Maximum Degree Greedy sera contrainte de retourner le pire cas. Pour cela, il suffit de construire un graphe Anti-MDG sans effectuer la seconde étape, c'est-à-dire celle où on ajoute les sommets de degré 2. La figure 1.14 donne un exemple d'un tel graphe pour $k = 6$.

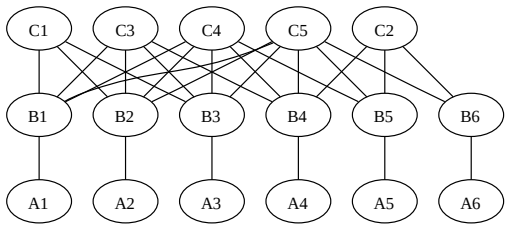


Fig. 1.14 – Exemple de graphe pour lequel Edge Deletion atteint son rapport d'approximation en pire cas

Dans cette classe de graphe, on montre avec les mêmes arguments que pour les graphes Anti-MDG que la taille de la solution optimale est k.

Reprenons le mécanisme de construction des graphes Anti-MDG décrit précédemment, en omettant les sommets de C de degré 2, et observons le degré des différents sommets du graphe, étape par étape. Lors de la première étape, nous créons un couplage entre les sommets de A et de B.

La deuxième étape consiste à rajouter $\lfloor \frac{k}{3} \rfloor$ sommets de degré 3 dans C. Tous les sommets de C ont donc un degré de 3. Les sommets de A ont un degré de 1, et les sommets de B ont (au plus) un degré de 2. Lorsque nous allons effectuer l'étape suivante en rajoutant $\lfloor \frac{k}{4} \rfloor$ sommets de degré 4 dans C, nous allons augmenter de 1 (au plus) le degré des sommets de B. Les sommets de B auront donc, au plus, un degré de 3. Comme les sommets que nous venons de rajouter dans C ont un degré de 4, l'ensemble C conserve les sommets de degré maximum du graphe. Ainsi, d'étape en étape, les sommets de degré maximum seront toujours dans C.

Observons maintenant le comportement de Maximum Degree Greedy sur le graphe ainsi construit. Tous les sommets de l'ensemble B sont de degré au plus $k-1$ car ils sont, au plus, reliés à un et un seul sommet de C de degré i avec $i \in \{3..k\}$ ainsi qu'à un sommet de A. Par construction, le sommet de degré maximum se trouve dans C et il est de degré k. Maximum Degree Greedy sera donc contraint de le sélectionner. Cette sélection va diminuer de 1 le degré de tous les sommets de l'ensemble B qui sont alors de degré $k-2$. À ce moment là, on se retrouve dans l'étape précédente de la construction, et nous avons vu que d'étape en étape, l'ensemble C contient tous les sommets de degré maximum. Maximum Degree Greedy va donc être contraint de sélectionner tous les sommets de C, repassant par chacune des étapes de construction dans l'ordre inverse. Lorsqu'il n'y a plus de sommet dans l'ensemble C, Maximum Degree Greedy va devoir couvrir chaque arête du couplage de la première étape et va donc retourner une solution de taille $\left(k + \lfloor \frac{k}{3} \rfloor + \lfloor \frac{k}{4} \rfloor + \ldots + 1\right) \approx \left(H(k) - \frac{1}{3}\right) \cdot k = \left(H(\Delta) - \frac{1}{3}\right) \cdot |OPT|$, avec Δ le degré maximum du graphe et H la série harmonique. Sur ce type de graphes, Maximum Degree Greedy possède donc un rapport d'approximation en pire cas et en moyenne de $H(\Delta)$ (en ordre de grandeur).

1.3 Bilan

Nous avons montré au moyen de deux algorithmes d'approximation pour le problème du vertex cover que l'évaluation en pire cas ne permet pas de capturer l'intégralité des comportements d'un algorithme sur les différentes classes de graphes. L'évaluation en pire cas est un outil précieux dans l'analyse des algorithmes mais elle n'est pas suffisante pour juger *a priori* des performances d'un algorithme sur une classe de graphes en particulier. Nous avons vu au moyen des étoiles qu'un algorithme peut être contraint de toujours retourner son pire cas (c'est le cas de Edge Deletion qui retourne toujours 2 sommets) tandis qu'un autre peut être contraint de toujours retourner la meilleure solution (c'est le cas de Maximum Degree Greedy qui ne retourne que le sommet central).

Le rapport d'approximation en pire cas est une vue macroscopique des performances en pire cas d'un algorithme. Cette vue peut être totalement décorrélée du rapport d'approximation en pire cas de ce même algorithme sur une classe de graphes en particulier, comme c'est le cas pour Maximum Degree Greedy sur les étoiles (son rapport d'approximation en pire cas est de $H(\Delta) = \sum_{i=1}^{\Delta} \frac{1}{i}$ tandis que son rapport d'approximation en pire cas sur les étoiles est de 1).

Le non-déterminisme, présent dans de nombreux algorithmes (par exemple dans le choix de l'arête à sélectionner pour Edge Deletion ou le choix du sommet de degré maximum pour Maximum Degree Greedy) peut influencer très fortement les performances d'un algorithme. Ainsi, sur la classe des graphes Anti-MDG spécialement conçue pour le piéger, Maximum Degree Greedy va obtenir d'excellents résultats en moyenne (l'espérance de son rapport d'approximation tend vers 1 lorsque la taille du graphe tend vers l'infini) tandis que sa pire exécution sur ces graphes

atteint son rapport d'approximation en pire cas. Ces bonnes performances proviennent du fait que les choix que Maximum Degree Greedy doit effectuer pour trouver la pire solution sont improbables, illustrant ainsi le fait que le rapport d'approximation en pire cas provient souvent d'une instance pathologique rare, doublée d'une exécution elle-même pathologique et rare.

Il convient toutefois de relativiser ces résultats. Le rapport d'approximation en pire cas reflète la réalité et les graphes pour lesquels un algorithme va retourner une solution qui atteint son rapport d'approximation en pire cas peuvent être légion, comme c'est le cas avec de nombreux graphes bipartis pour Edge Deletion (par exemple les arbres, les grilles et les hypercubes). De plus, nous avons modifié les graphes Anti-MDG de façon à piéger toute exécution de l'algorithme Maximum Degree Greedy.

Tous ces résultats nous montrent qu'on ne peut appréhender le comportement des algorithmes d'approximation uniquement en se focalisant sur le rapport d'approximation en pire cas et qu'il est donc nécessaire de recourir, en plus, à d'autres méthodes d'évaluation.

Chapitre 2

Comparaison de deux algorithmes de liste

Nous avons vu dans le chapitre précédent que l'évaluation en pire cas ne permet pas de juger *a priori* les performances globales d'un algorithme. Néanmoins, cette évaluation, lorsque l'on souhaite comparer deux algorithmes dont le rapport d'approximation en pire cas est différent, nous indique qu'il existe au moins un graphe et une exécution pour laquelle l'un des deux algorithmes va retourner une solution de meilleure qualité que l'autre (comme avec les algorithmes Edge Deletion et Maximum Degree Greedy par exemple). Au contraire, lorsque les deux algorithmes possèdent le même rapport d'approximation en pire cas, nous ne pouvons plus rien en déduire. Il convient donc de comparer les deux algorithmes de manière plus fine.

Dans ce chapitre, nous considérons des algorithmes d'approximation basés sur un ordre statique des sommets déterminé par leurs degrés. De tels algorithmes d'approximation sont appelés algorithmes de liste par analogie avec des heuristiques similaires pour l'ordonnancement de tâches.

Avis et Imamura ont proposé dans [AI07] un algorithme de liste (Sorted ListLeft) qui traite les sommets dans l'ordre décroissant de leur degré et dont le rapport d'approximation en pire cas est de $\frac{\sqrt{\Delta}}{2}+\frac{3}{2}$. Ils ont aussi montré que tout algorithme de liste possède un rapport d'approximation en pire cas d'au moins $\frac{\sqrt{\Delta}}{2}$, démontrant ainsi que Sorted ListLeft est très proche du meilleur rapport d'approximation en pire cas possible pour cette classe d'algorithmes.

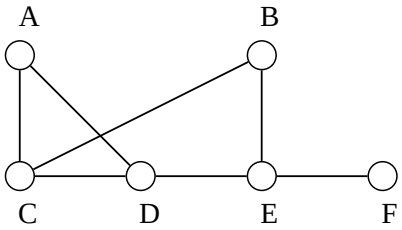
Nous proposons un algorithme de liste (Sorted ListRight) qui est « meilleur » que Sorted ListLeft. Plus précisément, nous montrons que pour toute liste $\mathcal{L}$ (et en particulier pour les listes triées par degrés décroissants), Sorted ListRight retourne une solution dont la taille est inférieure ou égale à celle retournée par Sorted ListLeft appliqué sur la même liste $\mathcal{L}$. De plus, nous exhibons une classe de graphes pour lesquels la différence de taille entre les deux solutions peut être arbitrairement élevée. Nous verrons aussi que ces deux algorithmes peuvent facilement s'adapter à un environnement distribué et qu'ils possèdent tous les deux de bonnes propriétés dans ce contexte.

Définition 6 (Algorithme de liste) *Un algorithme de liste parcourt les sommets du graphe un par un et dans un ordre déterminé à l'avance (une liste). Lorsque l'algorithme considère un sommet, il prend une décision irrévocable à son sujet (soit le sommet fait partie de la solution, soit il n'en fait pas partie).*

Notation 7 *Soit $\mathcal{L}$ une liste quelconque (c'est-à-dire une permutation quelconque des sommets du*

graphe). Pour tout sommet $u \in \mathcal{L}$, on note $N^R(u)$ (resp. $N^L(u)$) l'ensemble des voisins droits *(resp.* gauches*) de u, c'est-à-dire l'ensemble des voisins de u qui se trouvent à droite (resp. à gauche) de u dans $\mathcal{L}$. On dit que le sommet u est* monotone gauche *si $N^L(u) \neq \emptyset$ et $N^R(u) = \emptyset$ (u ne possède que des voisins gauches). On note $LM(\mathcal{L})$ l'ensemble des sommets monotones gauches.*

Exemple 8 *Considérons le graphe suivant et $\mathcal{L} = DECABF$ une liste associée. On a :*



- *C est à gauche de F dans $\mathcal{L}$.*
- *A est à droite de E dans $\mathcal{L}$.*
- *B est un voisin droit de E dans $\mathcal{L}$.*
- *D est un voisin gauche de C dans $\mathcal{L}$.*
- $N^R(C) = \{A, B\}$
- $N^L(B) = \{E, C\}$
- $N^L(D) = \emptyset$ *et* $N^R(F) = \emptyset$

2.1 Présentation des deux algorithmes de liste

Dans cette partie, nous présentons les deux algorithmes de liste étudiés dans ce chapitre. Ces deux algorithmes n'utilisent que des listes triées par degrés décroissants et nous allons voir que leur fonctionnement est très similaire.

Le premier algorithme, proposé par Avis et Imamura dans [AI07], est le suivant :

Algorithme 3 : SORTED LISTLEFT

Données : Un graphe G et une liste associée $\mathcal{L}$ triée par degrés décroissants
Sortie : Une couverture des sommets de G
$C \leftarrow \emptyset$;
Parcourir la liste $\mathcal{L}$ de la gauche vers la droite. Soit u le sommet courant.
si *u possède un voisin droit qui n'est pas dans C* **alors**
  $C \leftarrow C \cup \{u\}$
finsi
retourner C;

Cet algorithme possède un rapport d'approximation en pire cas de $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$ (voir [AI07]). Clairement, les solutions retournées par SORTED LISTLEFT ne sont pas forcément minimales pour l'inclusion. Il suffit de considérer le graphe 2.1 et la liste associée $\mathcal{L} = \{1, 2, 3, 4\}$ pour s'en rendre compte. L'algorithme va sélectionner les sommets 1, 2, et 3, ce qui n'est pas minimal pour l'inclusion car les sommets 1 et 3 forment à eux seuls un vertex cover.

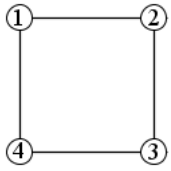


Fig. 2.1 – Exemple de graphe pour lequel l'algorithme Sorted ListLeft peut ne pas retourner une solution minimale pour l'inclusion.

Le corps de l'algorithme Sorted ListRight est identique à celui de Sorted ListLeft mis à part le sens de parcours de la liste. Nous verrons que cette simple différence va modifier le comportement de l'algorithme en lui interdisant la construction de solutions qui ne soient pas minimales pour l'inclusion.

Algorithme 4 : Sorted ListRight

Données : Un graphe G et une liste associée $\mathcal{L}$ triée par degrés décroissants
Sortie : Une couverture des sommets de G
$C \leftarrow \emptyset$;
Parcourir la liste $\mathcal{L}$ de la droite vers la gauche. Soit u le sommet courant.
si *u possède un voisin droit qui n'est pas dans C* **alors**
| $C \leftarrow C \cup \{u\}$
finsi
retourner C;

Lemme 9 Sorted ListRight *retourne une couverture des sommets de G qui est minimale pour l'inclusion.*

Preuve. Soient $e = uv$ une arête de G et C la solution courante. Supposons, sans perte de généralité que u se trouve à gauche de v dans $\mathcal{L} = \ldots u \ldots v \ldots$. Lorsque Sorted ListRight considère le sommet u : si v est déjà dans C alors l'arête e est couverte (au moins par v) ; sinon, u possède un voisin droit (v) qui n'est pas dans C donc u est ajouté dans C ; dans ce cas, e est couvert par u. C est donc une couverture des sommets de G. De plus, comme C ne peut contenir un sommet u et tout son voisinage (un sommet u ne peut être ajouté à la solution uniquement s'il possède un voisin droit qui n'est pas dans C), C est minimal pour l'inclusion. ■

2.2 L'algorithme Sorted ListRight est meilleur que l'algorithme Sorted ListLeft

Les travaux de cette section portant sur la comparaison des deux algorithmes de liste, ainsi que ceux de la suivante portant sur les graphes *ProW*, ont fait l'objet d'une publication en 2008 dans le journal Information Processing Letters, volume 107 ([DL08]).

Dans cette partie, nous montrons que Sorted ListRight retourne des solutions de meilleure qualité que Sorted ListLeft et nous verrons les différences entre leurs deux mécanismes de

sélection, ce qui nous permettra de préciser quels sont les sommets sélectionnés par chacun des deux algorithmes. Le résultat principal de cette partie est le suivant :

Théorème 10 *Pour toute liste,* Sorted ListRight *retourne une couverture de sommets dont la taille est inférieure ou égale à celle retournée par* Sorted ListLeft.

Preuve. Un sommet est dit monotone gauche s'il possède *au moins* un voisin gauche et *aucun* voisin droit. Soit $\mathcal{L}$ une liste quelconque. On note $LM(\mathcal{L})$ l'ensemble des sommets monotones gauche de $\mathcal{L}$ et $ISOL$ l'ensemble des sommets isolés (c'est-à-dire l'ensemble des sommets de degré 0). Soit $C_{SLR}(\mathcal{L})$ (resp. $C_{SLL}(\mathcal{L})$) l'ensemble des sommets retournés par Sorted ListRight (resp. Sorted ListLeft) lorsqu'il s'exécute sur la liste $\mathcal{L}$.

Il est facile de voir que $C_{SLR}(\mathcal{L})$ ne contient aucun sommet de $ISOL$, ni aucun sommet de $LM(\mathcal{L})$ car une condition nécessaire pour qu'un sommet soit ajouté dans $C_{SLR}(\mathcal{L})$ est qu'il possède au moins un voisin droit, ce qui n'est ni le cas des sommets de $ISOL$ ni le cas des sommets de $LM(\mathcal{L})$.

Soit $W = \mathcal{L} - C_{SLR}(\mathcal{L}) - ISOL - LM(\mathcal{L})$. Par définition, W ne contient aucun sommet de $C_{SLR}(\mathcal{L})$, de $ISOL$ ou de $LM(\mathcal{L})$. Nous avons vu que $C_{SLR}(\mathcal{L})$ ne contient aucun sommet de $ISOL$, ni aucun sommet de $LM(\mathcal{L})$, et par définition, $LM(\mathcal{L})$ ne contient aucun sommet de $ISOL$. Les ensembles $W, C_{SLR}(\mathcal{L}), ISOL$ et $LM(\mathcal{L})$ forment donc une partition de $\mathcal{L}$.

Pour montrer le théorème, nous allons montrer que $C_{SLL}(\mathcal{L}) = C_{SLR}(\mathcal{L}) \cup W$.

Clairement, $C_{SLL}(\mathcal{L})$ ne contient aucun sommet de $ISOL$ car une condition nécessaire pour qu'un sommet soit ajouté dans $C_{SLL}(\mathcal{L})$ est qu'il possède au moins un voisin droit. Soit u un sommet de $LM(\mathcal{L})$. Comme Sorted ListLeft parcourt la liste *de la gauche vers la droite*, lorsque u est considéré, tous ses voisins (qui sont tous des voisins gauches) ont déjà été parcourus et ajoutés dans C la solution courante (car lorsqu'ils ont été considérés, u était un voisin droit qui n'était pas encore dans C). Ainsi, u ne possède pas de voisin qui ne soit pas déjà dans C et donc u *n'est pas* ajouté dans C (et il ne sera pas ajouté plus tard car toute décision est irrévocable). Les arguments précédents montrent que $C_{SLL}(\mathcal{L})$ ne contient aucun sommet de $ISOL \cup LM(\mathcal{L})$.

Considérons maintenant les autres sommets de la partition de $\mathcal{L}$: l'ensemble $C_{SLR}(\mathcal{L}) \cup W$. Tout sommet u dans cet ensemble possède au moins un voisin droit w. Lorsque u est considéré, w n'est pas encore dans C puisqu'il n'a pas encore été parcouru ; ainsi Sorted ListLeft va ajouter u dans C.

Tous les arguments précédents montrent que $C_{SLL}(\mathcal{L}) = C_{SLR}(\mathcal{L}) \cup W$. Comme $C_{SLR}(\mathcal{L}) \subseteq C_{SLR}(\mathcal{L}) \cup W$, on a $C_{SLR}(\mathcal{L}) \subseteq C_{SLL}(\mathcal{L})$ et donc $|C_{SLR}(\mathcal{L})| \leq |C_{SLL}(\mathcal{L})|$, ce qui prouve le théorème. ■

Le théorème 10 montre que pour toute liste, Sorted ListRight domine Sorted ListLeft. C'est vrai pour toute liste et donc en particulier pour toutes les listes triées par degrés décroissants. Ce résultat nous permet d'obtenir le rapport d'approximation en pire cas de Sorted ListRight. En effet, notre algorithme ne peut pas retourner de solution dont la taille est supérieure à celle retournée par Sorted ListLeft, donc le rapport d'approximation en pire cas de Sorted ListRight est majoré par celui de Sorted ListLeft, soit $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$. Évidemment, si on souhaite obtenir la meilleure qualité de solution possible en utilisant un algorithme de liste, il conviendra donc d'utiliser notre algorithme mais nous verrons dans la section 2.4 que l'algorithme de Avis et Imamura possède certains avantages.

Pour aller plus loin. Dans la preuve du théorème 10, nous avons entraperçu la différence entre le mécanisme de sélection de SORTED LISTRIGHT et celui de SORTED LISTLEFT en fonction des positions des sommets du graphe dans la liste associée. Cependant, nous avons considéré la solution retournée par SORTED LISTRIGHT sans pour autant connaître la façon dont cette solution a été calculée. Nous allons maintenant analyser de manière plus fine le mécanisme de sélection des sommets utilisé par SORTED LISTRIGHT. Pour cela, nous allons partitionner les sommets en fonction de leurs relations de voisinage dans la liste.

Étant donnée une liste $\mathcal{L}$, on définit les ensembles W_i et B_i de la manière suivante :

$$B_1 = N^L(LM(\mathcal{L})) \qquad \text{et} \qquad W_1 = \{u : N^R(u) \subseteq B_1\}$$

Pour tout $i > 1$,

$$B_i = N^L(W_{i-1}) \qquad \text{et} \qquad W_i = \{u : N^R(u) \subseteq \bigcup_{j=1}^{i} B_j\}$$

B_i est l'ensemble des sommets qui possèdent au moins un voisin droit dans W_{i-1} (ou dans $LM(\mathcal{L})$ si $i = 1$). W_i est l'ensemble des sommets qui ont tous leurs voisins droits dans les ensembles B_j avec $j \leq i$. La figure 2.2 donne un exemple de graphe G avec une liste associée $\mathcal{L}$ ainsi qu'une partition de l'ensemble des sommets en sous-ensembles $LM(\mathcal{L}), W_i$ et B_i.

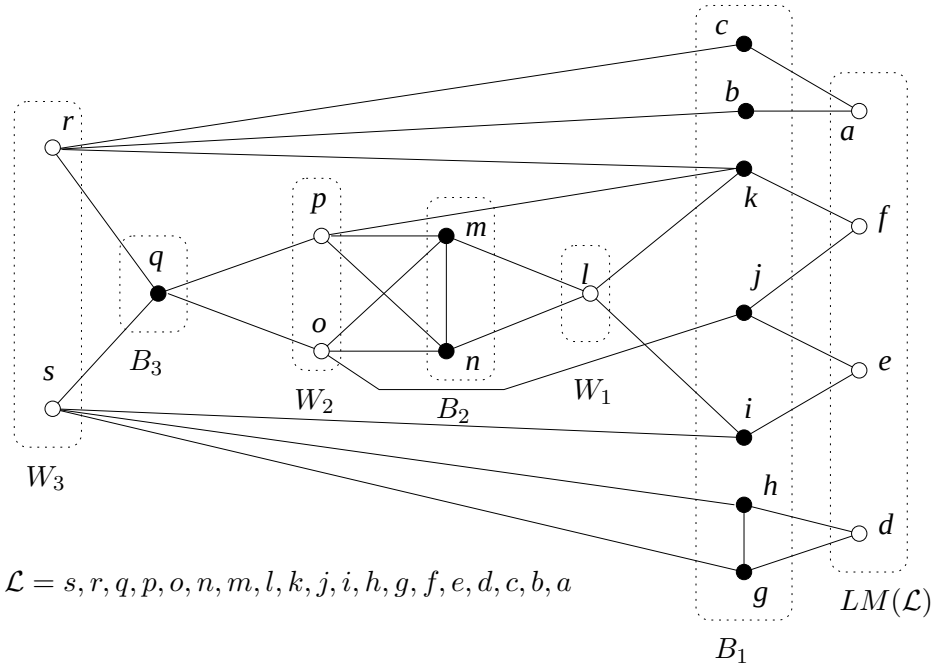


FIG. 2.2 – Un graphe, une liste associée $\mathcal{L}$ et les ensembles $LM(\mathcal{L})$, W_i et B_i construits à partir de $\mathcal{L}$

Nous avons déjà vu que SORTED LISTRIGHT ne retourne aucun sommet de $LM(\mathcal{L})$ (voir la preuve du théorème 10). Nous allons maintenant montrer que SORTED LISTRIGHT ne retourne que les sommets des ensembles B_i :

Lemme 11 *Pour toute liste $\mathcal{L}$, on a :*

$$\bigcup_{i\geq 1} B_i \subseteq C_{SLR}(\mathcal{L}) \qquad et \qquad \bigcup_{i\geq 1} W_i \cap C_{SLR}(\mathcal{L}) = \emptyset$$

Preuve. Soient $\mathcal{L}$ une liste quelconque et C la solution courante. Par définition des algorithmes de liste, on a $C \subseteq C_{SLR}(\mathcal{L})$ car un choix est irrévocable. Par induction sur i, on a :

$$B_i \subseteq C_{SLR}(\mathcal{L}) \qquad \text{et} \qquad W_i \cap C_{SLR}(\mathcal{L}) = \emptyset$$

Montrons la propriété pour $i = 1$.

Soit $u \in B_1$. Cela signifie que u possède au moins un voisin droit w dans $LM(\mathcal{L})$. Comme nous avons montré que $w \notin C_{SLR}(\mathcal{L})$ et comme w est considéré par Sorted ListRight avant u (w est un voisin droit de u), cela signifie que le voisinage de w (incluant u) est ajouté dans C. Comme la décision d'ajouter u dans C est irrévocable, il apparaît forcément dans la solution finale. Ainsi, $u \in C_{SLR}(\mathcal{L})$ et nous en déduisons que $B_1 \subseteq C_{SLR}(\mathcal{L})$.

Soit $u \in W_1$. Par définition, tous les voisins droits d'un sommet $u \in W_1$ (l'ensemble $N^R(u)$) se trouvent dans B_1. Comme nous avons montré que $B_1 \subseteq C_{SLR}(\mathcal{L})$, on obtient que $N^R(u) \subseteq C_{SLR}(\mathcal{L})$. Sorted ListRight ne va donc pas ajouter u dans C. Cette décision étant irrévocable, on a $u \notin C_{SLR}(\mathcal{L})$. Et nous obtenons que $W_1 \cap C_{SLR}(\mathcal{L}) = \emptyset$.

La propriété est vérifiée au rang $i = 1$. Supposons cette propriété vraie du rang 1 au rang $i-1$ et considérons les ensembles B_i et W_i.

Soit $u \in B_i$. Cela signifie que u possède au moins un voisin droit w dans W_{i-1}. Par hypothèse d'induction, $w \notin C_{SLR}(\mathcal{L})$ et comme w a déjà été considéré par Sorted ListRight, cela signifie que le voisinage de w (incluant u) est placé dans C. Ainsi, nous obtenons que $B_i \subseteq C_{SLR}(\mathcal{L})$.

Soit $u \in W_i$. Par définition, tous les voisins droits du sommet $u \in W_i$ (l'ensemble $N^R(u)$) appartiennent à un ensemble B_j ($j \leq i$).

Par hypothèse d'induction, $B_j \subseteq C_{SLR}(\mathcal{L})$ ($j \leq i$) et donc $N^R(u) \subseteq C_{SLR}(\mathcal{L})$. Nous pouvons utiliser exactement les mêmes arguments que pour W_1 et conclure que $W_i \cap C_{SLR}(\mathcal{L}) = \emptyset$. ■

Ordre relatif des sommets dans une liste. Le résultat précédent nous montre qu'en fait, ce n'est pas la position d'un sommet dans la liste en soit qui fait qu'un sommet sera ou non sélectionné par l'un ou l'autre des deux algorithmes, mais l'ordre relatif de ce sommet par rapport aux sommets de son voisinage. Soit A un ensemble ($LM(\mathcal{L})$, B_i ou W_i, avec $i \geq 1$), u et v deux sommets de A. L'ordre relatif de u et v dans $\mathcal{L}$ ($\mathcal{L} = \ldots u \ldots v \ldots$ ou $\mathcal{L} = \ldots v \ldots u \ldots$) n'a pas d'influence sur la solution retournée par Sorted ListLeft et Sorted ListRight. Plus précisément, les sommets u et v peuvent se trouver n'importe où dans la liste du moment qu'ils respectent l'ordre relatif suivant :

$$N^L(u) - A \ldots u \ldots N^R(u) - A \tag{2.1}$$
$$N^L(v) - A \ldots v \ldots N^R(v) - A \tag{2.2}$$

Comme u et v sont dans le même ensemble A :

- Si $A = LM(\mathcal{L})$, alors $N^R(u) = N^R(v) = \emptyset$. Sorted ListLeft et Sorted ListRight ne vont sélectionner ni u ni v.
- Si $A = B_i$ avec $i \geq 1$ alors u (resp. v) possède un voisin droit non sélectionné dans LM si $i = 1$ ou W_{i-1} si $i > 1$ (ce voisin est donc différent de v (resp. u)), donc u (resp. v) est sélectionné par Sorted ListLeft et Sorted ListRight.

– Si $A = W_i$ avec $(i \geq 1)$ alors $N^R(u) \neq \emptyset$, $N^R(v) \neq \emptyset$ et $N^R(u) \cup N^R(v) \in B_{i-1}$ donc Sorted ListRight ne sélectionnera ni u ni v tandis que Sorted ListLeft sélectionnera u et v.

Ainsi, lorsqu'on applique l'un des deux algorithmes Sorted ListRight ou Sorted ListLeft sur des listes différentes mais qui respectent les mêmes contraintes 2.1 et 2.2, nous obtiendrons la même solution, et cela sans tenir compte de la position de u dans ces listes par rapport à v, et sans tenir compte d'une éventuelle relation de voisinage entre les deux sommets.

Dans la preuve du théorème 10, nous avons partitionné l'ensemble des sommets en 4 sous-ensembles : $LM(\mathcal{L}), ISOL, C_{SLR}(\mathcal{L})$ et W. Nous avons montré que ni Sorted ListLeft ni Sorted ListRight ne retournent de sommet de $LM(\mathcal{L}) \cup ISOL$. Nous avons prouvé que Sorted ListLeft retourne tous les sommets sélectionnés par Sorted ListRight (c'est-à-dire $C_{SLR}(\mathcal{L})$) plus les sommets de l'ensemble W et nous venons de caractériser les sommets présents dans $C_{SLR}(\mathcal{L})$ en exhibant une méthode pour construire de façon ensembliste $C_{SLR}(\mathcal{L})$. On peut toutefois s'interroger sur l'ensemble W. Comme cet ensemble contient uniquement les sommets que Sorted ListLeft retourne en plus de la solution retournée par Sorted ListRight, la différence de qualité des solutions calculées par les deux algorithmes va dépendre de la taille de cet ensemble W. Nous allons montrer que $|W|$ peut être arbitrairement grand, illustrant ainsi le fait que l'on peut construire des graphes pour lesquels Sorted ListRight va retourner des solutions dont la qualité est nettement meilleure que celle des solutions retournées par Sorted ListLeft.

2.3 Les graphes *ProW*

Dans cette partie, nous exhibons une classe de graphes telle que pour toute liste associée, la taille de l'ensemble W présenté dans la section 2.2 est grande (d'où leur nom). Ainsi, pour toute liste triée par degrés décroissants, les performances de Sorted ListRight seront bonnes tandis que celles de Sorted ListLeft seront mauvaises. Plus précisément, on a :

Théorème 12 *Pour tout entier $N > 2$, il existe un graphe G tel que pour toute liste $\mathcal{L}$ associée à G et triée par degrés décroissants, l'algorithme* Sorted ListRight *retourne une solution optimale OPT tandis que l'algorithme* Sorted ListLeft *retourne une solution dont la taille est $\frac{N+1}{2} \cdot |OPT|$.*

Preuve. Soit $N > 2$ un entier. Nous allons construire un graphe G tel que pour toute liste $\mathcal{L}$ associée à G et triée par degrés décroissants, on a $|C_{SLL}(\mathcal{L})| = \frac{N+1}{2} \cdot |C_{SLR}(\mathcal{L})|$ et $C_{SLR}(\mathcal{L})$ est une solution optimale.

Le graphe G contient $N(N+3)$ sommets, répartis dans 5 ensembles disjoints V_1, F_1, V_2, F_2 et U. Les ensembles V_1, F_1, V_2 et F_2 contiennent N sommets chacun et l'ensemble U contient $N(N-1)$ sommets.

Les sommets de G sont reliés de la façon suivante :

– Chaque sommet de V_1 (resp. V_2) est connecté à un sommet différent de F_1 (resp. F_2).
– Chaque sommet de V_2 est connecté à tous les sommets de U.
– Chaque sommet de V_1 est connecté à $N-1$ sommets différents de U.

Soit u un sommet de G :

– Si $u \in F_1 \cup F_2$, $d(u) = 1$ (les sommets de $F_1 \cup F_2$ sont des feuilles).
– Si $u \in V_1$, $d(u) = N$ (u est connecté à un sommet de F_1 et $N-1$ sommets de U).
– Si $u \in U$, $d(u) = N+1$ (u est connecté à chacun des N sommets de V_2 et un sommet de V_1).

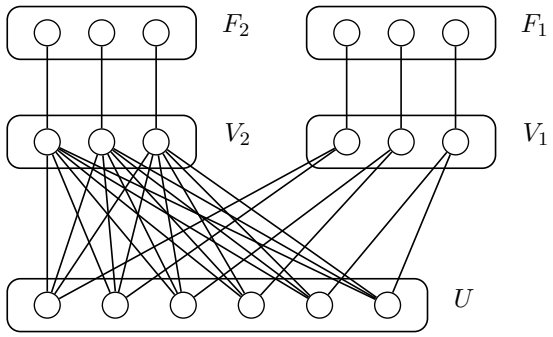


FIG. 2.3 – Exemple de graphe $ProW$ pour $N = 3$.

– Si $u \in V_2$, $d(u) = N(N-1)+1$ (u est connecté à chaque sommet de U plus un sommet de F_2).

Toute liste $\mathcal{L}$ associée à G et triée par degrés décroissants contient donc en premier les sommets de V_2, suivis par les sommets de U, puis ceux de V_1, et enfin les sommets de $F_1 \cup F_2$.

Étant donnée une telle liste $\mathcal{L}$, SORTED LISTLEFT va tout d'abord considérer les sommets de V_2 et va les ajouter dans la solution car tous les sommets de V_2 possèdent dans F_1 un voisin droit qui n'est pas encore dans la solution. Ensuite, SORTED LISTLEFT va regarder les sommets de U et va tous les ajouter dans sa solution. En effet, chaque sommet de U possède un voisin droit dans V_1 qui n'est pas encore sélectionné. Arrive le tour des sommets de V_1 qui sont aussi placés dans la solution car ils possèdent tous un voisin droit non encore sélectionné dans F_1. Enfin, SORTED LISTLEFT considère les sommets de $F_1 \cup F_2$. Aucun sommet de cet ensemble ne sera ajouté à la solution car tous leurs voisins sont déjà sélectionnés. Ainsi, SORTED LISTLEFT sélectionne tous les sommets de V_2, U et V_1 et on a $|C_{SLL}(\mathcal{L})| = N(N+1)$.

Étant donné la même liste $\mathcal{L}$, SORTED LISTRIGHT (qui parcourt $\mathcal{L}$ des degrés minimums vers les degrés maximums) commence par les sommets de $F_1 \cup F_2$ et ne les ajoute pas à la solution courante C car ils ne possèdent pas de voisin droit. Ensuite, les sommets de V_1 sont parcourus et sont tous placés dans C (car ils ont tous un voisin droit dans F_1 qui est non sélectionné). Les sommets de U sont ensuite considérés mais comme leurs seuls voisins droits sont les sommets de V_1 et que ces sommets sont déjà placés dans la solution, ils ne sont pas ajoutés dans la solution. Pour finir, les sommets de V_2 sont parcourus et sont ajoutés dans la solution car tous les sommets de V_2 possèdent un voisin droit non sélectionné dans U. Nous obtenons que $C_{SLR}(\mathcal{L}) = V_1 \cup V_2$ et $|C_{SLR}(\mathcal{L})| = 2N$.

Soit $C_{ED} = V_1 \cup F_1 \cup V_2 \cup F_2$ une solution que l'algorithme EDGE DELETION peut retourner (il s'agit d'un couplage entre les sommets de $V_1 \cup V_2$ et les sommets de $F_1 \cup F_2$). C_{ED} est de taille $4N$. Comme EDGE DELETION est $2-$approché et que $|C_{SLR}(\mathcal{L})| = 2N$, cela signifie que $C_{SLR}(\mathcal{L})$ est une solution optimale. De plus, $|C_{SLL}(\mathcal{L})| = N(N+1) = \frac{2N(N+1)}{2} = |C_{SLR}(\mathcal{L})|\frac{N+1}{2}$, ce qui prouve le résultat. ■

2.4 Deux algorithmes facilement distribuables

Nous avons vu que SORTED LISTRIGHT est plus performant que SORTED LISTLEFT dans le sens où SORTED LISTLEFT retourne toujours une taille de solution supérieure ou égale à celle retournée

par SORTED LISTRIGHT et nous avons montré que la différence de taille entre les deux solutions peut être aussi grande que l'on veut pour certaines instances particulières. Cependant, l'algorithme SORTED LISTLEFT possède quelques bonnes propriétés qui, dans certaines circonstances, peuvent nous faire préférer son utilisation plutôt que celle de SORTED LISTRIGHT. Plus précisément, nous allons voir que les deux algorithmes se distribuent très facilement et que SORTED LISTLEFT utilise moins de messages que SORTED LISTRIGHT et que le nombre de rounds de communications utilisés par SORTED LISTLEFT pour calculer la solution est lui aussi nettement inférieur.

Un système distribué est une machine parallèle à la différence qu'il y a plusieurs entités de calculs autonomes distantes géographiquement. La différence principale avec la programmation d'architectures parallèles repose donc sur le fait que la distance entre les entités de calcul est plus importante (d'une pièce, d'un campus, d'une ville, d'un pays ou au niveau planétaire) et nécessite donc forcément l'utilisation d'un réseau de communication (type Internet). Dans un système distribué, les nœuds n'ont qu'une vision locale du système, c'est-à-dire qu'ils ne connaissent que leur voisinage dans le graphe (noté $N(u)$ pour un nœud u). Bien que de nombreux modèles existent, nous nous focalisons ici sur deux d'entre eux : les modèles synchrones et asynchrones.

Le modèle synchrone assure que chaque machine (ou processeur) possède une horloge locale propre. Ces horloges sont toutes incrémentées en même temps au cours de l'algorithme et marquent la même valeur. En l'absence de défaillance, tout message envoyé par un processus au temps t à un voisin est reçu et traité au temps $t+1$. Dans ce type de réseau, tous les processus commencent simultanément leur activité au signal d'un initiateur global. Cela revient à supposer l'existence d'une horloge globale au réseau qui synchronise cycle par cycle chaque événement et chaque action de tout processus. Sur une ligne de communication donnée, chaque processus ne peut envoyer qu'au plus un message à chaque cycle d'horloge et le délai d'acheminement d'un message est inférieur ou égal à cette unité de temps. Le cycle de calcul de chaque processus est donc le suivant :

1. Envoyer des messages vers 1 ou plusieurs voisins.
2. Recevoir des messages de 1 ou plusieurs voisins.
3. Effectuer le calcul local. Le calcul local est supposé prendre un temps négligeable devant l'étape 2.

Dans le modèle asynchrone au contraire, l'algorithme est dirigé par les événements de réception de messages car les processus ne peuvent avoir accès à une horloge globale pour prendre leur décision. Les messages envoyés d'un nœud v à un nœud u arrivent en un temps fini mais imprévisible. Par exemple, on ne peut pas déduire l'ordre d'arrivée des messages venant des voisins u_i de v d'après leur ordre d'émission.

Remarque : on supposera dans la suite qu'il n'y a aucune faute (pas de liens ou nœuds pouvant tomber en panne, pas de changement de topologie du graphe, ...) et que chaque processus possède une identité unique.

Pour le problème du vertex cover dans un environnement distribué, on sait calculer une solution $2-$approchée en calculant un couplage maximal et en ajoutant les sommets du couplage dans la solution. Un tel couplage peut être calculé en $O(\log^4 n)$ rounds de communication en utilisant

l'algorithme de Hanckowiack et al. [HKP98] et en $O(\Delta + \log^* n)$ en utilisant l'algorithme de Panconesi and Rizzi [PR01]. On peut aussi trouver un couplage maximal en un nombre de rounds dont l'espérance est en $O(\log n)$ via l'algorithme probabiliste de Israeli et Itai [II86]. Bien que ces algorithmes soient présentés pour le modèle synchrone, on peut en déduire un algorithme asynchrone en effectuant quelques modifications mineures.

Un algorithme distribué est dit local si chaque nœud ne nécessite que la connaissance de son voisinage pour déterminer s'il fait partie ou non de la solution. Kuhn et al. [KMW04, KMW06] ont montré qu'il n'existe pas d'algorithme d'approximation à facteur constant qui soit local pour le problème du vertex cover dans les graphes en général (sans que le degré soit borné), et Polishchuk et Suomela [PS09] ont proposé un algorithme local 3−approché pour les graphes de degré borné.

Nous allons voir que les algorithmes Sorted ListLeft et Sorted ListRight peuvent facilement être transformés en algorithmes distribués asynchrones. Ces deux algorithmes, que l'on appellera SLL_{rep} et SLR_{rep} construisent exactement les mêmes couvertures de sommets que Sorted ListLeft et Sorted ListRight appliqués sur une certaine liste triée.

2.4.1 Description des algorithmes SLL_{rep} et SLR_{rep}

Dans cette partie, nous présentons les versions réparties des algorithmes Sorted ListRight et Sorted ListLeft pour le modèle asynchrone. Les deux algorithmes possèdent une première phase commune qui permet de déterminer pour chaque sommet l'ensemble de ses voisins droits et l'ensemble de ses voisins gauches. Une fois cette phase effectuée, SLL_{rep} est capable de déterminer localement quels sont les sommets faisant partie de la solution, tandis que SLR_{rep} va nécessiter l'envoi de messages, de proche en proche, afin que chaque sommet soit capable de déterminer s'il fait partie d'un ensemble B_i, $LM(\mathcal{L})$ ou W_i (ces ensembles ont été décrits en section 2.2). Nous montrons ensuite comment adapter ces algorithmes pour le modèle synchrone.

Remarque. Les algorithmes distribués présentés dans cette section n'ont besoin d'aucune connaissance sur le système (comme le nombre de machines ou encore le diamètre du réseau). Les identifiants doivent être deux à deux comparables et, par simplicité, on utilisera les opérateurs $<$ et $>$ pour les comparer.

Phase 1 (commune aux deux algorithmes) : création d'une liste triée locale

Au départ, tous les nœuds excepté un nombre quelconque (au moins un) sont endormis. Lorsqu'un nœud est réveillé, il envoie à tous ses voisins un message contenant son identité u et son degré $d(u)$ (c'est-à-dire le nombre de ses voisins). Si un nœud qui n'est pas réveillé reçoit un tel message, il se réveille. Par ce processus, tous les nœuds sont réveillés et connaissent l'identité et le degré de chacun de leurs voisins.

Lorsqu'un nœud a reçu un message de chacun de ses voisins, il construit deux ensembles :

$$N^R(u) = \{v \in N(u) : (d(v) < d(u)) \text{ ou } (d(v) = d(u) \text{ et } v < u)\}$$

$$N^L(u) = N(u) - N^R(u)$$

Plus précisément, le nœud u construit localement une partie d'une liste triée. Si un voisin v est de degré strictement inférieur (resp. strictement supérieur) que son propre degré $d(u)$, le voisin v

devient un voisin droit (resp. un voisin gauche) et de façon symétrique u devient un voisin gauche (resp. droit) de v. Si $d(v) = d(u)$ et $v < u$, le nœud u considère que le nœud v est un voisin droit (et de façon symétrique le nœud v considère que u est un voisin gauche).

Phase 2 de l'algorithme SLL_{rep}

Une fois que le nœud u a construit ces deux ensembles, si $N^R(u) = \emptyset$ il décide de ne pas se placer dans la solution C (en configurant un drapeau local), sinon il décide de s'ajouter à la solution puis il s'arrête. Lorsque tous les nœuds se sont arrêtés, la couverture de sommets est construite.

Phase 2 de l'algorithme SLR_{rep}

Une fois que le nœud u a construit ces deux ensembles, si $N^R(u) = \emptyset$ et $N^L(u) \neq \emptyset$ il décide de ne pas se placer dans la solution C (en configurant un drapeau local) et il envoie à tous ses voisins gauches un message $< NotInC >$ puis il s'arrête.

Lorsqu'un nœud v reçoit pour la première fois un message $< NotInC >$, il se place dans la solution C et envoie un message $< InC >$ à tous ses voisins gauches. Ensuite, quels que soient les messages qu'il recevra par la suite de la part de ses autres voisins droits, il ne fera rien et attendra d'avoir reçu un message de la part de tous ses voisins droits avant de s'arrêter.

Si un nœud u reçoit seulement des messages $< InC >$ de la part de tous ses voisins droits, il décide alors de ne pas se placer dans la solution C et il envoie un message $< NotInC >$ à tous ses voisins gauches avant de s'arrêter.

Lorsque tous les nœuds sont arrêtés, la couverture de sommets est construite.

Remarque : ces deux algorithmes fonctionnent aussi si le graphe n'est pas connexe (dans le cas de disparition de liens par exemple arrivant avant le début de l'exécution de l'algorithme) si au moins un nœud est réveillé dans chaque composante connexe, ce qui nous amène à considérer le modèle synchrone. En effet, dans ce modèle, tous les nœuds sont réveillés au même moment au signal d'un initiateur global, ce qui assure que chaque composante connexe possède un nœud réveillé. Les deux algorithmes s'adaptent trivialement au modèle synchrone car leur comportement est identique si ce n'est que les nœuds ne sont pas réveillés un par un lors de la phase commune, mais tous simultanément.

2.4.2 Évaluation des deux algorithmes répartis

Suivant que l'on se place dans le modèle synchrone ou asynchrone, l'évaluation des performances ne se fait pas de la même manière. Certes, on souhaite obtenir une solution dont la taille est la plus faible possible, mais au-delà de ce critère il convient de réduire au maximum le nombre de rounds de communications (de cycles d'horloge) afin d'obtenir une solution en un temps raisonnable. Le nombre de messages échangés doit lui aussi être le plus faible possible pour ne pas engorger le réseau ce qui diminuerait ses performances et entraînerait des risques d'erreurs. Dans un premier temps, nous verrons que les deux algorithmes SLL_{rep} et SLR_{rep} se comportent comme les algorithmes Sorted ListLeft et Sorted ListRight, garantissant ainsi une solution au plus $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}-$approchée, puis nous verrons que le nombre de messages échangés par les deux algorithmes est relativement faible. Enfin, nous montrerons que SLL_{rep} s'exécute en nombre de rounds de communication constant, contrairement à SLR_{rep}.

Comportement de SLL_{rep} et SLR_{rep}. L'exécution de nos deux algorithmes distribués est similaire à l'exécution des algorithmes Sorted ListLeft et Sorted ListRight, ce qui, par le théorème 10, garantit une solution au plus $\frac{\sqrt{\Delta}}{2}+\frac{3}{2}-$approchée mais aussi que la taille de la solution calculée par Sorted ListRight$_{rep}$ est inférieure ou égale à celle de SLL_{rep}. On note $\mathcal{L}$ la liste globale suivante : un nœud u est à gauche du nœud v si $d(u) > d(v)$ ou $d(u) = d(v)$ et $u > v$. Ce tri est un *ordre total* sur les nœuds (étant donné deux nœuds u et v, u est situé soit à gauche de v soit à droite dans $\mathcal{L}$ mais il n'y a aucune ambiguïté car les identifiants des sommets sont uniques et forment donc eux aussi un ordre total) et $\mathcal{L}$ est une liste triée (si $\mathcal{L} = \ldots, u, \ldots, v, \ldots$ alors $d(u) \geq d(v)$).

Durant l'exécution de nos algorithmes distribués, le nœud u va construire une « vue locale » de $\mathcal{L}$ (plus précisément les ensembles $N^L(u)$ et $N^R(u)$). Ensuite, les algorithmes distribués suivent le comportement de Sorted ListLeft et Sorted ListRight. Nous exploitons ici le fait que la liste $\mathcal{L}$ n'a pas besoin d'être traitée séquentiellement mais qu'elle peut au contraire être traitée en parallèle, en accord avec les ensembles B_i et W_i définis dans la partie précédente.

Nous allons maintenant évaluer le nombre de messages échangés par les deux algorithmes. Cette évaluation est valable à la fois pour le modèle synchrone et pour le modèle asynchrone.

Nombre de messages échangés. Nous avons vu dans la partie précédente que Sorted ListLeft a besoin de moins d'information que Sorted ListRight pour s'exécuter. Plus précisément, pour sélectionner un sommet il doit juste savoir si ce sommet possède un voisin droit dans la liste, tandis que Sorted ListRight doit savoir si tous ses voisins droits sont sélectionnés avant de décider de ne pas sélectionner un sommet. De façon non surprenante, cette différence de quantité d'information requise va se répercuter sur le nombre de messages échangés. Plus précisément :

Lemme 13 *Soit m le nombre d'arêtes du graphe G. L'algorithme SLL_{rep} (resp. SLR_{rep}) échange exactement $2m$ (resp. $3m$) messages.*

Preuve. Lors de la phase commune aux deux algorithmes (création d'une liste triée locale), chaque arête est traversée exactement une fois dans chaque direction par un message initial contenant l'identité et le degré. Cette étape génère exactement $2m$ messages.

Après cette phase commune, l'algorithme SLL_{rep} n'utilise pas d'autres messages, car toutes les informations pour qu'un sommet u effectue le choix de se placer (si $N^R(u) \neq \emptyset$) ou non (si $N^R(u) = \emptyset$) dans la solution sont déjà connues par le sommet u. Le nombre de messages échangés est donc $2m$.

Après la phase commune, l'algorithme SLR_{rep} utilise les messages $< InC >$ et $< NotInC >$. Un seul de ces deux messages va traverser une arête, et seulement « de la droite vers la gauche », ce qui va générer exactement m messages. Comme il n'y a pas d'autres messages, l'algorithme SLR_{rep} va donc échanger exactement $3m$ messages.

On peut noter que les $2m$ premiers messages contiennent seulement l'identité et un entier $d \leq n$ (codé sur $\lceil \log_2 n \rceil$ bits chacun). Les m derniers messages ne contiennent aucune donnée. ■

On peut voir les m messages supplémentaires de SLR_{rep} comme un moyen d'améliorer la solution. Il convient donc de choisir l'algorithme à utiliser en fonction de ce que l'on souhaite : réduire le nombre de messages ou tenter d'améliorer la qualité de la solution retournée.

Nous allons maintenant évaluer le nombre de rounds de communication consommés par les deux algorithmes. Cette évaluation n'a de sens que pour le modèle synchrone car dans notre modèle, nous n'avons pas fixé de borne sur le temps de transmission d'un message.

Nombre de rounds de communication. Nous allons voir que l'algorithme SLL_{rep}, qui envoie moins de messages que l'algorithme SLR_{rep}, s'exécute en un nombre de rounds de communications moins important que SLR_{rep}. Ce qui est notable toutefois, c'est que SLL_{rep}, contrairement à SLR_{rep}, est un algorithme local. En effet, SLL_{rep} envoie et reçoit des messages uniquement dans la phase commune. Dans le modèle synchrone, tous les nœuds se réveillent au même moment. Ils envoient donc leur degré à tous leurs voisins et reçoivent le degré de tous leurs voisins en 1 seul round de communication. Le temps de calcul nécessaire pour la création de la vue locale de la liste triée étant considéré comme négligeable par rapport au temps de réception des messages, on obtient que la phase commune se déroule en 1 seul round de communication, ce qui en fait un algorithme local. La seule façon d'obtenir un algorithme plus rapide serait de ne pas envoyer de messages afin de s'exécuter en 0 round de communication et seul l'algorithme trivial qui consiste à sélectionner tous les sommets en est capable*. On peut donc dire que SLL_{rep} est optimal en nombre de rounds de communication.

L'algorithme SLR_{rep} peut avoir besoin de $\Theta(n)$ rounds de communication, avec n le nombre de nœuds du réseau. Il suffit pour cela de considérer le cas d'un chemin. On peut très facilement trouver une liste telle que seuls les sommets du bord du chemin sont monotones gauche. Ces sommets vont envoyer un message à leurs voisins (soit à 2 sommets au total) qui vont eux même transmettre un message à leurs 2 voisins gauches et ainsi de suite. La phase 2 de SLR_{rep} peut donc s'exécuter en $\Theta(\frac{1}{2}n) = \Theta(n)$ rounds de communication. Plus précisément, le nombre de rounds de communication sera égal au nombre total d'ensembles B_i et W_i et LM.

Une perspective intéressante sera de déterminer le nombre moyen de rounds de communications que SLR_{rep} met pour s'exécuter. En effet, dans de nombreuses classes de graphes, le diamètre est faible (par exemple dans les graphes aléatoires de Erdös-Renyi et plus généralement dans les graphes petit monde), ce qui indique un nombre d'ensembles B_i et W_i relativement faible.

Quel algorithme choisir ? Pour répondre à cette question, il faut définir le critère que l'on souhaite optimiser. L'algorithme Sorted ListRight va retourner une solution dont la taille est inférieure ou égale à celle retournée par Sorted ListLeft. C'est donc cet algorithme que nous devons choisir si nous souhaitons mettre toutes les chances de notre côté pour obtenir une solution de meilleure qualité. D'un autre côté, si notre objectif est de réduire au maximum le nombre de messages échangés ou bien le temps de calcul de la solution (en terme de rounds de communication) il faudra alors préférer l'algorithme Sorted ListLeft.

2.5 Bilan

L'objectif de ce chapitre était de comparer deux algorithmes afin d'appréhender leurs forces et faiblesses tout en allant plus loin qu'une « simple » évaluation en pire cas. Nous avons étudié l'algorithme Sorted ListLeft qui est un algorithme de liste dont le rapport d'approximation en pire cas est de $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$ et nous avons proposé Sorted ListRight, un autre algorithme de liste qui,

*et on peut difficilement dire qu'il s'agit d'un algorithme raisonnable.

dans son énoncé, ressemble beaucoup à Sorted ListLeft. Nous avons montré que notre algorithme retourne une solution dont la taille est toujours inférieure ou égale à celle de Sorted ListLeft et qui est minimale pour l'inclusion, ce qui n'est pas toujours le cas de Sorted ListLeft. Nous avons aussi voulu savoir à quel point notre algorithme peut être meilleur. Pour cela, nous avons exhibé une classe de graphes telle que toute exécution de Sorted ListRight est optimale tandis que toute exécution de Sorted ListLeft retourne une solution de mauvaise qualité, illustrant le fait que notre algorithme peut être arbitrairement meilleur sur certaines classes de graphes. Le comportement de ces deux algorithmes les rend facilement distribuables tout en conservant la relation de dominance mise en lumière. Bien que la version distribuée de Sorted ListLeft soit moins bonne que celle de Sorted ListRight du point de vue de la qualité de la solution construite, nous avons montré qu'elle est plus efficace en terme de nombre de rounds de communications ainsi qu'en nombre de messages échangés.

Chapitre 3

Évaluation en moyenne de deux algorithmes de liste

Dans le chapitre précédent, nous avons comparé deux algorithmes de liste triée par degrés décroissants. Le fait que les listes soient triées permettait d'obtenir un rapport d'approximation de $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$ ce qui est relativement bon étant donné qu'aucun algorithme de liste ne peut assurer un rapport d'approximation inférieur à $\frac{\sqrt{\Delta}}{2}$. Cependant, ce même tri permettait de piéger les deux algorithmes, comme avec les graphes ProW et ceux de Avis et Imamura [AI07] par exemple.

Dans ce chapitre, nous relâchons la contrainte qui imposait que les sommets d'une liste devaient être tries par degrés décroissants et nous évaluons en moyenne les performances des deux algorithmes de liste.

Les travaux présentés dans les deux premières sections de ce chapitre ont fait l'objet d'une publication en 2009 dans le journal Information Processing Letters, volume 109 ([BDL09]).

3.1 Sorted ListRight et Sorted ListLeft face au passé et au futur

Les algorithmes Sorted ListLeft et Sorted ListRight parcourent la liste une seule fois et prennent la décision (irrévocable) d'inclure (ou non) un sommet dans la solution au moment où ce sommet est considéré. Cependant, lorsqu'un sommet u est considéré par Sorted ListRight, la décision d'inclure (ou non) ce sommet dans la solution dépend seulement de l'état des sommets qui ont déjà été parcourus. Ainsi, si on présente le graphe comme un « flux de sommets » s'écoulant dans le même sens que les algorithmes considèrent les sommets (de la droite vers la gauche de la liste pour Sorted ListRight et de la gauche vers la droite pour Sorted ListLeft), Sorted ListRight va prendre sa décision uniquement avec l'information déjà connue/ apparue dans le flux (c'est-à-dire le sommet courant u et les sommets à droite de u dans la liste) car il n'a pas besoin de savoir si u aura un « futur » voisin dans le flux (c'est-à-dire un voisin à gauche de u dans la liste). D'un autre coté Sorted ListLeft a besoin de savoir si le sommet courant u aura un voisin dans la partie future du flux (c'est-à-dire un voisin à droite de u dans la liste). Plus précisément, Sorted ListLeft n'a besoin que des données futures et n'utilise à aucun moment les informations déjà connues/apparues dans le flux. Ces deux comportements ont chacun leurs avantages :

– L'algorithme Sorted ListLeft n'a pas besoin de mémoire, car ses choix ne dépendent que des sommets qui vont être révélés. Cet algorithme peut donc être utilisé dans le cadre

d'applications ayant une quantité limitée de mémoire.
– Le comportement de l'algorithme SORTED LISTRIGHT est un comportement dit « online ». Un algorithme online n'a pas besoin de connaître l'instance complète dès le départ : elle est révélée au fur et à mesure, sommet par sommet (par exemple) et lorsqu'un sommet est révélé, l'algorithme online doit prendre une décision irrévocable le concernant.

Un algorithme online. Demange et Paschos ont proposé l'algorithme online suivant dans [DP05] : au départ, le graphe G est totalement inconnu et la solution courante C est vide. À chaque étape, un sommet u est révélé avec les arêtes incidentes aux sommets précédemment révélés. Lorsque u est révélé, l'algorithme online le sélectionne (c'est-à-dire qu'il place u dans C) si et seulement si u possède un voisin déjà révélé et qui n'est pas dans C.
Il a été montré dans [DP05] que cet algorithme online retourne une solution qui est minimale pour l'inclusion et qu'il possède un rapport d'approximation en pire cas de Δ, avec Δ le degré maximum du graphe. Notons que ce rapport d'approximation est atteint lorsque le graphe est une étoile, c'est-à-dire un arbre composé d'un sommet central et de $n-1$ feuilles connectées directement au centre. Si le centre est révélé en premier, il n'est pas ajouté dans la solution courante et chaque feuille révélée par la suite sera ajoutée dans C. A la fin, on a $|C| = n-1 = \Delta$ alors que la couverture optimale, de taille 1, n'est composée que du sommet central, nous amenant ainsi au rapport d'approximation en pire cas annoncé.

On peut remarquer que SORTED LISTRIGHT se comporte exactement comme l'algorithme online si on considère que l'ordre des sommets dans la liste représente l'ordre inverse de révélation des sommets (c'est-à-dire que les sommets sont révélés de la droite de la liste vers la gauche).

Lemme 14 *Si les sommets sont révélés dans le même ordre que* SORTED LISTRIGHT *considère les sommets, alors les algorithmes online et* SORTED LISTRIGHT *retournent la même solution.*

Preuve. Considérons un graphe et une liste qui lui est associée. L'algorithme online va considérer les sommets dans le même ordre que LISTRIGHT (c'est-à-dire que les sommets sont révélés dans l'ordre inverse de la liste, soit de la droite vers la gauche). Aucun des deux algorithmes ne sélectionne le premier sommet. Supposons que les deux algorithmes sélectionnent les mêmes sommets jusqu'au $(n-1)-$ème sommet révélé/considéré, et considérons u, le $n-$ième sommet révélé/considéré.
– Si u est sélectionné par l'algorithme online, cela signifie qu'il possède un voisin v déjà révélé non sélectionné. Par hypothèse, LISTRIGHT n'a pas sélectionné v, qui se trouve à droite dans la liste. LISTRIGHT va donc lui aussi sélectionner le sommet u.
– Si l'algorithme online ne sélectionne pas le sommet u, cela signifie que tous ses voisins déjà révélés se trouvent déja dans la solution. Par hypothèse, cela signifie que LISTRIGHT a aussi sélectionné tous les voisins de u qui se trouvent à droite dans la liste et il n'ajoutera donc pas le sommet u dans la solution.

Les deux algorithmes retournent donc la même solution lorsque les sommets sont révélés/considérés dans le même ordre. ■

Ce résultat nous permet d'appliquer l'analyse effectuée dans le chapitre 2 à l'algorithme online :

Corollaire 15 *Lorsque les sommets sont révélés dans l'ordre croissant des degrés, alors l'algorithme online (qui possède un rapport de compétitivité de Δ, voir [DP05]) retourne un vertex cover au plus $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$-approché.*

Ceci améliore grandement ses performances en pire cas. Plus généralement, comme ces deux algorithmes retournent exactement les mêmes solutions, tous les résultats obtenus pour l'un des deux

s'appliquent aussi à l'autre algorithme du moment que les contraintes sont les mêmes comme l'ordre d'apparition/de considération des sommets par exemple.

Remarque. Le fait que deux algorithmes retournent exactement la même solution ne signifie pas que ces algorithmes sont les mêmes. Par exemple, deux algorithmes de tri peuvent toujours retourner la même solution sans pour autant fonctionner de la même façon. Dans le cas présent, Sorted ListRight et online, bien qu'ayant un fonctionnement très proche, ne sont pas basés sur le même modèle : l'un est online tandis que l'autre, de liste, connait l'instance dans son intégralité dès le début de l'exécution. Il s'agit donc bien de deux algorithmes différents.

Relâchement de la contrainte de tri des listes. Le tri des listes par degrés décroissants permet de piéger les deux algorithmes de liste de telle sorte qu'ils retournent, pour toute exécution, une solution de relativement mauvaise qualité (voir les graphes ProW présentés en 2.3 pour l'algorithme Sorted ListLeft par exemple et les graphes de Avis et Imamura [AI07] pour Sorted ListRight). De plus, demander que les listes soient triées au départ est une contrainte relativement forte étant donné que le but des algorithmes de liste est de traiter des instances de la façon la plus rapide possible, en une seule passe, et une phase de tri, même rapide et préalable à l'algorithme, diminue leur intérêt. Pour ces raisons, nous avons décidé de relâcher la contrainte qui imposait que Sorted ListRight et Sorted ListLeft soient appliqués uniquement sur des listes triées par degrés décroissants. Les résultats que nous obtiendrons pour l'algorithme Sorted ListRight seront aussi valables pour l'algorithme online de Demange et Paschos (voir lemme 17), sans se restreindre à des ordres de révélations particuliers.

Pour éviter les confusions, nous noterons ListRight (resp. ListLeft) l'algorithme Sorted ListRight (resp. Sorted ListLeft) lorsqu'on l'applique à une liste quelconque, qui n'est donc pas forcément triée par degrés décroissants.

Exemple. Nous allons voir, sur un exemple simple, quelle va être l'influence de la suppression de cette contrainte sur les performances des deux algorithmes. On considère S_n, une étoile à n sommets. Dans toutes les listes de S_n triées par degrés décroissants, le sommet central (de degré $\Delta = n-1$) se situe à l'extrémité gauche de la liste et les $n-1$ feuilles (de degré 1) à sa droite dans un ordre quelconque. Sorted ListLeft considère le sommet central en premier et va le sélectionner car il possède exactement $n-1$ voisins droits non sélectionnés. Ensuite, il va parcourir la liste sans jamais placer d'autre sommet dans la solution car toutes les feuilles ne possèdent aucun voisin à droite dans la liste. Sorted ListRight quant à lui, considère en premier les $n-1$ feuilles mais n'en sélectionne aucune car leur seul voisin (le sommet central) se trouve tout à gauche dans la liste. Le dernier sommet considéré étant le sommet central dont tous les voisins droits sont non sélectionnés, l'algorithme va l'ajouter à la solution. Ainsi, dans le cas particulier des étoiles, les deux algorithmes Sorted ListLeft et Sorted ListRight retournent une solution de taille 1 (ce qui est optimal).

On considère maintenant que les listes ne sont plus triées et qu'elles sont toutes équiprobables. Évaluer en pire cas nos algorithmes de liste sur les étoiles à n sommets revient donc à trouver la liste qui maximise la taille de la solution pour chacun des deux algorithmes. Il suffit pour cela de considérer une liste dans laquelle le sommet central de l'étoile se trouve à l'extrême droite. ListLeft va considérer toutes les feuilles une à une et va, à chaque fois, les sélectionner car elles possèdent toutes un voisin droit non sélectionné, à savoir le sommet central. Sur cette même liste, l'algorithme

ListRight va retourner la même solution. En effet, le premier sommet considéré par l'algorithme sera le sommet central, qui ne possède pas de voisin droit et qui ne sera donc pas sélectionné. Ensuite, toutes les feuilles seront sélectionnées par l'algorithme car elles possèdent un voisin droit non sélectionné, le sommet central. Ainsi, les deux algorithmes peuvent retourner une solution de taille $n-1 = \Delta$ alors que la solution optimale est de taille 1, ce qui nous donne un rapport d'approximation en pire cas de Δ, ce qui est bien plus mauvais que la $\left(\frac{\sqrt{\Delta}}{2} + \frac{3}{2}\right)$−approximation obtenue avec les listes triées. La suppression de la contrainte de tri des listes n'est pas, *a priori*, une bonne idée si on souhaite évaluer en pire cas les algorithmes ListRight et ListLeft. Plus précisément, il est connu (voir [DP05]) qu'un algorithme minimal pour l'inclusion (comme ListRight par exemple) ne peut pas retourner de solution dont le rapport d'approximation est supérieur à Δ. ListRight possède donc le pire rapport d'approximation en pire cas possible. Quant à l'algorithme ListLeft (qui n'est pas minimal pour l'inclusion), nous avons montré que son rapport d'approximation est au moins de Δ, ce qui correspond presque au pire rapport d'approximation en pire cas possible. En effet, aucun algorithme ignorant les sommets isolés ne peut retourner de solution dont le rapport d'approximation en pire cas est supérieur à $\Delta + 1$. Pour s'en convaincre, il suffit de considérer une solution optimale OPT. Par définition, la taille de son voisinage est au plus de $\Delta|OPT|$ et un algorithme qui sélectionne tous les sommets non isolés va donc retourner au plus $|OPT| + \Delta|OPT|$ sommets ce qui correspond à une solution $\Delta + 1$ approchée.

Cependant, on peut remarquer que dans le cas de l'algorithme ListRight, seules les listes pour lesquelles le sommet central se trouve tout à droite permettent de retourner cette « mauvaise » solution, toutes les autres listes retournant une solution optimale. Imaginons que le premier sommet considéré (et qui n'est donc pas sélectionné) par ListRight soit une feuille. ListRight va ensuite considérer les autres sommets un par un. Soit u le sommet courant.

- Si u est une feuille et que le sommet central est un voisin gauche de u. Le sommet u n'est pas sélectionné car il ne possède pas de voisin droit.
- Si u est le sommet central, il est forcément sélectionné par ListRight car le premier sommet considéré et non sélectionné appartient à son voisinage droit.
- Si u est une feuille et que le sommet central est un voisin droit de u, alors le sommet u n'est pas sélectionné car il ne possède pas de voisin droit non sélectionné.

Nous pouvons en déduire l'espérance de la taille de la solution retournée par ListRight, en considérant que toutes les listes sont équiprobables.
Pour rappel, dans le cas d'une variable discrète X pouvant prendre un nombre fini n de valeurs réelles : $x_1, x_2, \ldots, x_n$ avec les probabilités $p_1, p_2, \ldots, p_n$ alors l'espérance de X vaut $\mathbb{E}(X) = \sum_{i=1}^{n} x_i p_i$.
Soit S_n une étoile à n sommets. Il y a exactement $\frac{(n-1)!}{n!} = \frac{1}{n}$ listes telles que le sommet central se trouve tout à droite. On en déduit qu'avec probabilité $\frac{1}{n}$, l'algorithme ListRight va retourner une solution de taille $n-1$ et de taille 1 avec probabilité $\frac{n-1}{n}$. D'où

$$\mathbb{E}(\text{ListRight}(S_n)) = \frac{1}{n}(n-1) + \frac{n-1}{n}1 = 2 - \frac{2}{n}.$$

Concernant l'algorithme ListLeft, on obtient de la même façon que précédemment que la probabilité que le sommet central se trouve en $i$ème position est de $\frac{1}{n}$. Lorsque le sommet central se trouve en $i$ème position (avec $i \neq n$), ListLeft va sélectionner tous les sommets précédents (soit $i-1$ sommets) car ils possèdent tous comme voisin droit le sommet central, puis le sommet central

car il possède encore au moins une feuille comme voisin droit (le dernier sommet de la liste). Aucun autre sommet ne sera sélectionné car ils se trouvent tous à droite du sommet central dans la liste et ne possèdent donc pas de voisin droit. ListLeft sélectionne donc exactement i sommets. Lorsque le sommet central se trouve en dernière position de la liste, on retrouve le comportement en pire cas et l'algorithme va donc sélectionner $n-1$ sommets. Ainsi, on a :

$$\mathbb{E}(\textsc{ListLeft}(S_n)) = \frac{1}{n}(n-1) + \frac{1}{n}\sum_{i=1}^{n-1} i = \frac{n+1}{2} - \frac{1}{n}.$$

Dans cet exemple, nous venons de voir que si nous relâchons la contrainte de tri des listes, nous dégradons du même coup la qualité de la solution pouvant être retournée en pire cas par chacun des deux algorithmes de liste. L'évaluation de l'espérance de la taille des solutions retournées, lorsque l'on considère que toutes les listes sont équiprobables nous a permis de montrer que ListLeft peut être arbitrairement mauvais en moyenne tandis que ListRight va retourner des solutions dont la qualité est bonne. Dans ce qui suit, nous allons voir que pour ListRight, les étoiles constituent le pire cas en moyenne (c'est-à-dire que les étoiles forment la classe de graphes pour laquelle les performances moyennes de ListRight sont les plus mauvaises) et que pour tout graphe, ses performances moyennes sont bonnes, contrairement à ce que laisse supposer son évaluation en pire cas.

Hypothèse 16 *Dans la suite de la première partie de cette thèse, chaque fois qu'un choix pourra être effectué, il sera fait de manière équiprobable parmi l'ensemble des choix possibles.*

Cela signifie par exemple, que toutes les listes ont autant de chance d'être considérées. Ainsi, pour un graphe de taille n, il y a exactement $n!$ listes possibles (il s'agit du nombre de permutations de n éléments) et on considère que toutes les listes ont la même probabilité d'être considérées soit $\frac{1}{n!}$. Cela signifie aussi que dans le cas de l'algorithme Edge Deletion, le choix d'une arête se fera de manière équiprobable parmi l'ensemble des arêtes disponibles ou encore que l'algorithme Maximum Degree Greedy va choisir de façon équiprobable un sommet de degré maximum parmi tous les sommets de degré maximum.

3.2 LR est 2-approché en moyenne

Nous avons vu dans l'exemple précédent que l'algorithme ListRight possède de bonnes performances moyennes sur les étoiles. Dans cette partie, nous allons montrer que ListRight possède de bonnes performances moyennes pour tout graphe. Pour cela, nous allons donner un nouvel algorithme et, comme pour l'algorithme online, nous allons montrer qu'il se comporte de la même façon que ListRight du moment que les sommets sont considérés dans le même ordre.

L'algorithme *LRglouton***.** Au début, aucun sommet n'est sélectionné. Les sommets sont examinés un par un dans n'importe quel ordre donné. Soit u le sommet courant (celui que l'on examine). Si u n'est pas sélectionné, alors on sélectionne tout son voisinage. Lorsque tous les sommets ont été considérés, on retourne tous les sommets sélectionnés.

Lemme 17 *Si l'algorithme LRglouton considère les sommets dans le même ordre que* ListRight*, alors les deux algorithmes retournent la même solution.*

Preuve. Considérons n'importe quel ordre donné des sommets. Soit u le sommet courant. Si u est le premier sommet examiné alors u n'est pas sélectionné, que ce soit par ListRight ou par *LRglouton*. En effet, ListRight ne sélectionne jamais le premier sommet de la liste car il ne possède pas de voisin droit et *LRglouton* ne sélectionne pas le premier sommet qu'il considère mais il sélectionne son voisinage.

Supposons maintenant que les deux algorithmes sélectionnent exactement les mêmes sommets jusqu'au sommet courant. Soit u le prochain sommet examiné. Soit *LRglouton* sélectionne u, soit il ne le sélectionne pas :

- Si u n'a pas été sélectionné par *LRglouton* lors d'une étape précédente, cela signifie que jusqu'à u aucun voisin de u n'a été examiné, ou que tous les voisins examinés de u ont été sélectionnés. Comme ListRight sélectionne exactement les mêmes sommets que *LRglouton* jusqu'à u, cela signifie que u ne possède pas de voisin déjà révélé qui ne soit pas sélectionné. Dans cette situation, aucun des deux algorithmes ne sélectionne u.
- Si u a été sélectionné par *LRglouton* lors d'une étape précédente, cela signifie que jusqu'à u au moins un voisin de u (déjà révélé) n'a pas été sélectionné. Comme c'est la même chose pour ListRight par hypothèse, ListRight sélectionne u. Les deux algorithmes sélectionnent u.

Cela prouve que lorsque les deux algorithmes examinent les sommets dans le même ordre, ListRight sélectionne u si et seulement si *LRglouton* sélectionne u. Ainsi *LRglouton* se comporte de la même façon que ListRight. ■

Étant donné que *LRglouton* et ListRight se comportent de la même façon, nous considèrerons les deux algorithmes indistinctement dans la preuve du théorème suivant :

Théorème 18 *Pour tout graphe G à n sommets* $\mathbb{E}(\text{ListRight}(G)) \leq \left(2 - \frac{2}{n}\right) opt(G)$ *et cette borne est atteinte.*

Preuve. Avant de prouver le théorème par induction sur $opt(G)$, nous allons montrer que l'ensemble $ISOL$ qui contient les sommets isolés (c'est-à-dire l'ensemble des sommets sans voisins) n'est pas important et qu'il peut être omis lors de l'analyse.

Tout d'abord, il est facile de voir que $opt(G) = opt(G - ISOL)$. De plus, on a $\mathbb{E}(\text{ListRight}(G))) = \mathbb{E}(\text{ListRight}(G - ISOL)))$. Soit $k = |ISOL|$ et $\mathbb{L}(G)$ l'ensemble des ordres de révélations des sommets de G. On a $|\mathbb{L}(G)| = n!$ et $|\mathbb{L}(G - ISOL)| = (n-k)!$. On note $O(L)$ l'ordre relatif des sommets dans $L \in \mathbb{L}(G)$. Si $L \in \mathbb{L}(G - ISOL)$, il y a exactement $\frac{n!}{(n-k)!}$ ordres de révélations $L_i \in \mathbb{L}(G)$ tels que $O(L_i - ISOL) = O(L)$ et comme on a clairement $\text{ListRight}(L_i) = \text{ListRight}(L_i - ISOL)$, nous obtenons que :

$$\mathbb{E}(\text{ListRight}(G)) = \frac{1}{n!} \sum_{L \in \mathbb{L}(G)} \text{ListRight}(L) = \frac{1}{n!} \sum_{L \in \mathbb{L}(G - ISOL)} \frac{n!}{(n-k)!} \text{ListRight}(L) =$$

$$\frac{1}{(n-k)!} \sum_{L \in \mathbb{L}(G - ISOL)} \text{ListRight}(L) = \mathbb{E}(\text{ListRight}(G - ISOL))$$

Cas de base de l'induction. Soit $G = (V, E)$ un graphe à $n \geq 1$ sommets tel que $opt(G) = 0$. Comme $opt(G) = 0$, cela signifie que G ne contient aucune arête. L'algorithme ListRight ne sélectionnera donc aucun sommet car il faudrait pour cela qu'un sommet possède au moins un

voisin droit et cela n'est possible que s'il existe au moins une arête et nous en déduisons que $\mathbb{E}(\textsc{ListRight}(G)) = 0$. Ainsi $0 = \mathbb{E}(\textsc{ListRight}(G)) \leq \left(2 - \frac{2}{n}\right) opt(G) = 0$. Ce qui prouve le cas de base.

Étant donné un sommet $v \in V$, on définit le graphe $G_{\setminus v}$ comme le graphe induit par $V - (N(v) \cup \{v\})$ (c'est-à-dire G moins le sommet v et son voisinage $N(v)$). Notons que dans *LRglouton*, la version équivalente de ListRight, si le sommet v est examiné en premier, alors $N(v)$ est sélectionné et à partir de ce moment, ni v ni $N(v)$ n'auront d'impact sur la sélection des futurs sommets. Ainsi, pour l'analyse du comportement de l'algorithme après avoir examiné v, nous devons seulement nous focaliser sur $G_{\setminus v}$ (sur lequel nous appliquons l'hypothèse d'induction).

Hypothèse d'induction. Soit $k \geq 1$; on suppose que $\mathbb{E}(\textsc{ListRight}(G')) \leq (2 - \frac{2}{n'}) opt(G')$ pour tout graphe G' avec n' sommets tels que $opt(G') \leq k - 1$.

Maintenant, considérons un graphe G avec n sommets et $opt(G) = k$. Nous allons montrer que $\mathbb{E}(\textsc{ListRight}(G)) \leq \left(2 - \frac{2}{n}\right) opt(G)$.

Avec la remarque faite avant l'induction, nous pouvons considérer que G ne contient pas de sommet isolé. Soit H une couverture optimale de G ($|H| = opt(G)$). Soit I l'ensemble $V - H$. Il est facile de voir que I est un ensemble de sommets indépendants (il n'y a aucune arête entre deux sommets de I sinon l'un des deux sommets devrait être dans la solution et ce sommet ne se trouverait donc pas dans I mais dans H) et chaque sommet de I possède au moins un voisin dans H (sinon il serait isolé, et on considère qu'il n'y a pas de sommet isolé).

Soit $d_I(v)$ (resp. $d_H(v)$) le nombre de sommets dans $N(v) \cap I$ (resp. $N(v) \cap H$). Donc, $d(v) = d_I(v) + d_H(v)$.

Soit v un sommet de G. On a, $H \cap V(G_{\setminus v})$ est toujours une couverture de $G_{\setminus v}$. Ainsi donc,

$$opt(G_{\setminus v}) \leq |H \cap V(G_{\setminus v})| \leq opt(G) - d_H(v) - \mathbb{I}_{v \in H} \tag{3.1}$$

ou $\mathbb{I}_{v \in H}$ vaut 1 si $v \in H$ et 0 si $v \in I$.

Notons que pour tout $v \in V$, l'hypothèse d'induction peut être appliquée sur $G_{\setminus v}$ (en effet, si $v \in H$, $\mathbb{I}_{v \in H} = 1$ et par (3.1), $opt(G_{\setminus v}) \leq opt(G) - 1$; si $v \in I$, $d_H(v) \geq 1$ et par (3.1), $opt(G_{\setminus v}) \leq opt(G) - 1$). Écrivons maintenant l'induction :

$$\begin{aligned}
\mathbb{E}(\textsc{ListRight}(G)) &= \sum_{v \in V} \mathbb{E}(\textsc{ListRight}(G) | v \text{ est révélé en premier }) \mathbb{P}(v \text{ est révélé en premier }) \\
&= \frac{1}{n} \sum_{v \in V} \left(d(v) + \mathbb{E}(\textsc{ListRight}(G_{\setminus v}))\right) \\
&\leq \frac{1}{n} \sum_{v \in V} \left(d(v) + 2opt(G_{\setminus v})\right) \text{ par induction et comme } 2 - \frac{2}{n} \leq 2 \\
&\leq \frac{1}{n} \sum_{v \in V} \left(d(v) + 2(opt(G) - d_H(v) - \mathbb{I}_{v \in H})\right) \text{ par (3.1)} \\
&= 2opt(G) + \frac{1}{n} \sum_{v \in V} (d_I(v) - d_H(v)) - \frac{2}{n} \sum_{v \in V} \mathbb{I}_{v \in H} \qquad (3.2)
\end{aligned}$$

Comme I est un ensemble indépendant, $\sum_{v\in I} d_I(v) = 0$. De plus, si on note $e_{H,I}$ le nombre d'arêtes entre H et I,

$$\sum_{v\in I} d_H(v) = e_{H,I} = \sum_{v\in H} d_I(v)$$

Pour cette raison,

$$\begin{aligned}\sum_{v\in V(G)} (d_I(v) - d_H(v)) &= \sum_{v\in I} d_I(v) - \sum_{v\in I} d_H(v) + \sum_{v\in H} d_I(v) - \sum_{v\in H} d_H(v) \\ &= -\sum_{v\in H} d_H(v) \leq 0\end{aligned}$$

En reportant dans (3.2) on obtient

$$\mathbb{E}(\textsc{ListRight}(G)) \leq 2opt(G) - \frac{2}{n}\sum_{v\in V} \mathbb{I}_{v\in H}$$

et comme $\sum_{v\in V} \mathbb{I}_{v\in H} = opt(G)$, on obtient le résultat.

Nous avons vu dans l'exemple de la section 3.1 que la borne est atteinte pour une étoile S_n à n sommets car $\mathbb{E}(\textsc{ListRight}(S_n)) = 2 - \frac{2}{n}$. ■

Un corollaire immédiat du théorème 18 est que pour tout graphe, l'espérance du rapport d'approximation de ListRight est bornée par 2, une constante. Nous avons exhibé une classe de graphes sur laquelle l'espérance du rapport d'approximation de ListLeft est de $\frac{n+1}{2} - \frac{1}{n}$. Comme le théorème 10 est vrai pour toute liste, il s'applique aussi sur les algorithmes ListRight et ListLeft. Ces différents arguments montrent que lorsque la contrainte de tri est relâchée, non seulement ListRight retourne toujours une solution dont la taille est inférieure ou égale à celle retournée par ListLeft, mais la différence de performance en moyenne peut être aussi importante que l'on souhaite.

3.3 Comparaison de ListRight et ListLeft sur les graphes d'Erdös-Renyi

Nous avons vu que ListLeft peut être arbitrairement mauvais en moyenne (sur les étoiles par exemple), tandis que ListRight est au plus 2−approché en moyenne. Dans la continuité de notre étude, nous souhaitons savoir si les mauvaises performances de ListLeft sur certaines classes de graphes ont un impact important si l'on considère l'ensemble des graphes possibles ou si au contraire ses performances tendent à égaler celles de ListRight. Pour cela, nous allons comparer les performances des algorithmes ListLeft et ListRight sur l'ensemble des graphes à n sommets. Plus précisément, nous allons calculer l'espérance de la taille des solutions retournées par ces deux algorithmes sachant seulement qu'on va tirer au hasard (et de façon équiprobable) un graphe à n sommets parmi l'ensemble de tous les graphes à n sommets. La seule information connue est la

taille du graphe n. Pour mener à bien cette étude, nous allons devoir utiliser les graphes d'Erdős-Rényi. Un graphe d'Erdős-Rényi (noté $G(n,p)$ est un graphe aléatoire qui se construit à partir d'un ensemble de n sommets initialement non connectés et d'un réel $0 \leq p \leq 1$ appelé probabilité de connexion. Pour chacune des $\frac{n\cdot(n-1)}{2}$ paires de sommets $\{i,j\}$, on crée l'arête (i,j) avec probabilité p (on laisse donc i et j non connectés avec probabilité $1-p$).

Tirer au hasard et de façon équiprobable un graphe à n sommets parmi l'ensemble de tous les graphes à n sommets revient à générer un graphe $G(n,\frac{1}{2})$ (voir [Bol01]). Ainsi, nous cherchons à calculer $\mathbb{E}(A(G(n,\frac{1}{2})))$, avec A l'un de nos deux algorithmes. Pour cela, nous allons utiliser les séries génératrices qui sont un outil formel pour manipuler une distribution de probabilités et pour dériver facilement la moyenne et l'écart-type, par exemple. Cet outil devient fondamental lorsqu'on aborde l'analyse d'algorithmes. Par ce biais, on obtient le résultat suivant :

Théorème 19 *Soit $G \in G_{n,p}$, avec $0 \leq p \leq 1$, et $q = 1-p$. On a :*

$$\begin{aligned}
\mathbb{E}(\textsc{ListLeft}(G)) &= n - \frac{1-q^n}{1-q} \\
Var(\textsc{ListLeft}(G)) &= \frac{1-q^n}{1-q} - \frac{1-q^{2n}}{1-q^2} \\
\mathbb{E}(\textsc{ListRight}(G)) &= (n-1)p + \sum_{k=2}^{n-1} \binom{n-1}{k} (-q)^k \prod_{l=1}^{k-1}(1-q^l) \\
Var(\textsc{ListRight}(G)) &= (n-1)(n-2)(1-2q) + \\
&\quad 2\sum_{k=2}^{n-1} \binom{n-1}{k} (n-1-k)(-q)^k \prod_{l=1}^{k-1}(1-q^l) \left(n-2-k+\sum_{l=1}^{k-1} \frac{1}{1-q^l} \right)
\end{aligned}$$

La preuve de ce théorème se trouve en section 3.3.1.

Nous savons par le théorème 10 que ListRight retourne toujours une solution de taille inférieure ou égale à celle retournée par ListLeft, l'espérance de la taille des solutions retournées par ListRight est donc inférieure ou égale à celle des solutions retournées par ListLeft. Nous allons exploiter graphiquement le théorème 19 pour nous rendre compte des différences de performances des deux algorithmes lorsque l'on considère différentes tailles de graphes et probabilités.

Remarque. Étant habitué à considérer des classes de graphes particulières telles que les arbres, les grilles ou les hypercubes, il est toujours surprenant de voir que ces graphes sont statistiquement inexistants (c'est-à-dire que leur nombre par rapport au nombre total de graphes est insignifiant) et que la grande majorité des graphes se ressemblent. Ainsi, si on génère un graphe $G_{n,p}$, on a de grandes chances d'obtenir un graphe connexe dont le diamètre est faible et dont la distribution des degrés suit une loi binomiale. Un résultat de Bollobas [Bol01] indique que pour une probabilité d'arête p fixée, la taille d'un vertex cover minimum va se rapprocher de plus en plus de n lorsque l'on augmente la taille du graphe. Il est donc évident que l'écart de performance entre les deux algorithmes va se réduire au fur et à mesure que l'on va considérer des tailles de graphes de plus en plus importantes.

La figure 3.1 nous montre de quelle façon évoluent les espérances des tailles des solutions retournées par les deux algorithmes lorsque l'on fixe la taille du graphe à 100 et que l'on fait varier la probabilité d'existence d'une arête. On obtient un écart de l'espérance de la taille des solutions d'environ 12% lorque $p = 0.13$, 5% lorque $p = \frac{1}{2}$ et de 3% lorque $p = 0.7$. Des résultats similaires sont obtenus lorsque l'on change la taille du graphe, ce qui démontre que l'algorithme ListRight est d'autant plus efficace par rapport à ListLeft que la densité (en nombre d'arêtes) du graphe est faible.

La figure 3.2 nous montre le pourcentage d'amélioration de l'espérance de ListRight par rapport à celle de ListLeft lorsque l'on fixe p et que l'on fait varier la taille du graphe.

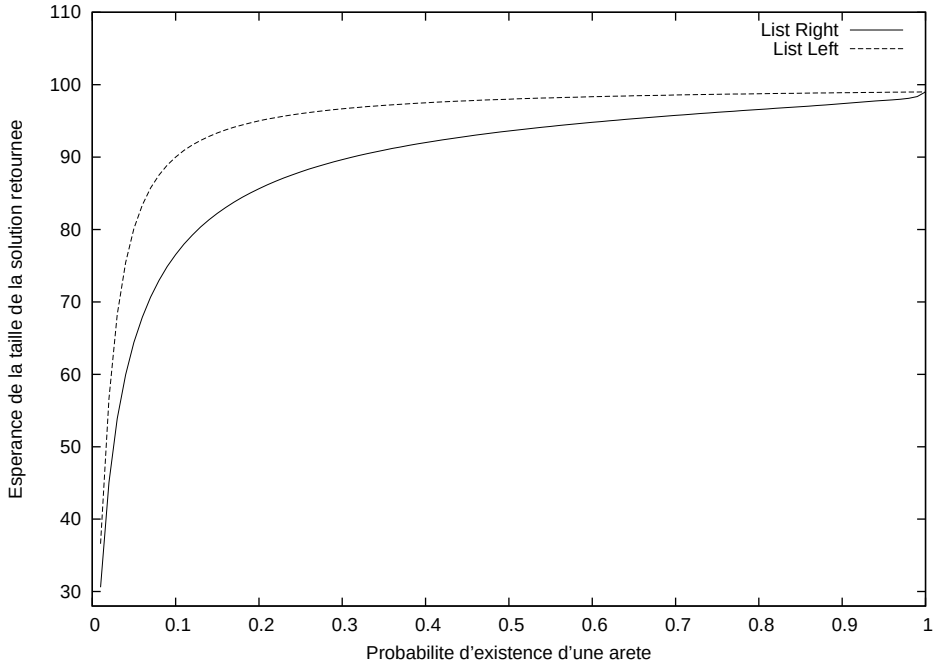


Fig. 3.1 – Comparaison de l'espérance des algorithmes ListLeft et ListRight sur les graphes $G_{n,p}$ lorsque $n = 100$ et p varie.

Calcul effectué pour un graphe G. Une diminution de t% se traduit par une multiplication de $1 - \frac{t}{100}$. Sachant que $\mathbb{E}(\textsc{ListRight}(G)) \leq \mathbb{E}(\textsc{ListLeft}(G))$ on sait que $\mathbb{E}(\textsc{ListRight}(G)) = \mathbb{E}(\textsc{ListLeft}(G))\left(1 - \frac{t}{100}\right)$ et on en déduit que le pourcentage de diminution t vaut $\left(1 - \frac{\mathbb{E}(\textsc{ListRight}(G))}{\mathbb{E}(\textsc{ListLeft}(G))}\right) \cdot 100$.

La figure 3.2 nous indique clairement que ListRight est d'autant plus efficace, par rapport à ListLeft, que la probabilité d'existence d'une arête est faible. L'espérance de la taille de la solution retournée par ListRight peut-être de 0.7% inferieure à celle de ListLeft pour $n = 500$ et $p = 0.9$ et jusqu'a 21% inferieure à celle de ListLeft pour $n = 50$ et $p = 0.1$. Les faibles écarts obtenus pour $n = 500$ sont tout de même significatifs car pour cette taille et un probabilité d'arête élevée, les graphes générés sont presques des graphes complets.

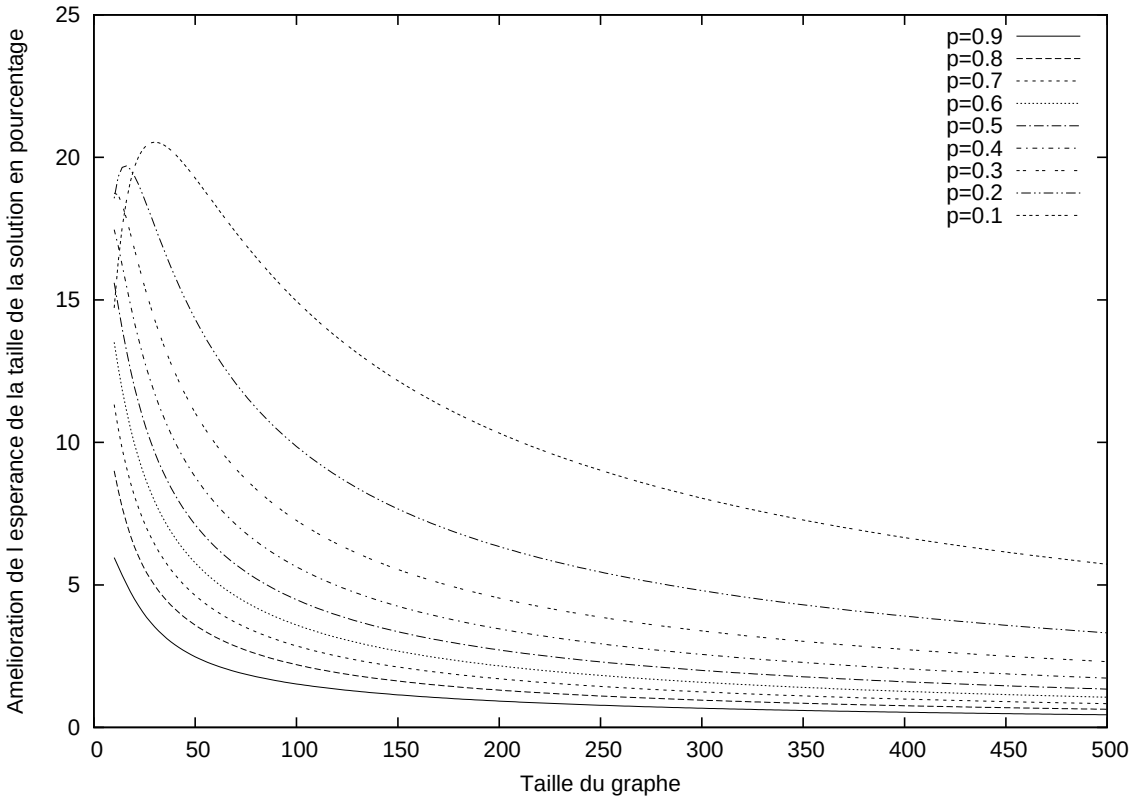


FIG. 3.2 – Pourcentage d'amélioration de l'espérance de ListRight par rapport à celle de ListLeft sur des graphes de Erdös-Renyi lorsque la taille du graphe va de 10 à 500 et pour différentes probabilités d'arêtes.

Ces résultats confirment le fait que l'algorithme ListRight possède un meilleur comportement que l'algorithme ListLeft sur les graphes aléatoires en général. Notre objectif était de savoir si les mauvaises performances de ListLeft sur certaines classes de graphes ont un impact important si l'on considère l'ensemble des graphes possibles ou si au contraire ses performances tendent à égaler celles de ListRight. Les figures 3.1 et 3.2 nous montrent que l'écart de performance entre les deux algorithmes est significative et n'est absolument pas négligeable.

3.3.1 Preuve du théorème 19

Espérance et variance de ListLeft

Soit S_m le nombre de sommets sélectionné par ListLeft lorsque m sommets ont été parcourus (ainsi, dans le cas particulier où $m = n$, avec n le nombre de sommets du graphe, on obtient que S_m est égal à la solution retournée par ListLeft). On note F_m la série génératrice de S_m, c'est-à-dire le polynôme définit par

$$F_m(X) = \sum_{k=0}^{m} \mathbb{P}(S_m = k) X^k$$

Lemme 20

$$\forall m \geq 1, F_m(X) = \prod_{k=1}^{m} \left(q^{n-k} + (1-q^{n-k})X\right)$$

Preuve. Nous prouvons le lemme par induction sur m. Le premier sommet n'est pas sélectionné uniquement s'il s'agit d'un sommet isolé. Cet événement arrive avec probabilité q^{n-1}. Ainsi, $F_1(X) = q^{n-1} + (1-q^{n-1})X$ et le résultat est donc vérifié pour $m = 1$. Nous supposons la formule du lemme 20 vraie jusqu'au rang $m-1$ et nous vérifions au rang m.

Pour $m \geq 2$, nous pouvons écrire l'égalité suivante pour tout $k \geq 0$:

$$\mathbb{P}(S_m = k) = \mathbb{P}(S_m = k|S_{m-1} = k)\mathbb{P}(S_{m-1} = k) + \mathbb{P}(S_m = k|S_{m-1} = k-1)\mathbb{P}(S_{m-1} = k-1) \quad (3.3)$$

$\mathbb{P}(S_m = k|S_{m-1} = k)$ étant la probabilité pour le sommet m de ne pas être sélectionné, sachant que k sommets ont déjà été sélectionnés auparavant. Ce sommet n'est pas sélectionné uniquement s'il ne possède pas de voisin droit, ce qui arrive avec probabilité q^{n-m}, et avec le même argument, on obtient que $\mathbb{P}(S_m = k|S_{m-1} = k-1) = 1 - q^{n-m}$. Ainsi, l'égalité 3.3 nous donne

$$\mathbb{P}(S_m = k) = q^{n-m}\mathbb{P}(S_{m-1} = k) + (1-q^{n-m})\mathbb{P}(S_{m-1} = k-1)$$

$$\Rightarrow \sum_{k=0}^{m} \mathbb{P}(S_m = k)X^k = q^{n-m}\sum_{k=0}^{m} \mathbb{P}(S_{m-1} = k)X^k + (1-q^{n-m})X\sum_{k=0}^{m} \mathbb{P}(S_{m-1} = k-1)X^{k-1}$$

Comme ListLeft ne peut pas sélectionner plus de sommets qu'il n'en a parcouru dans la liste, on a $\mathbb{P}(S_{m-1} = m) = 0$ et de la même manière, il ne peut pas retourner de solution de taille négative d'où $\mathbb{P}(S_{m-1} = -1) = 0$. On obtient donc que

$$\sum_{k=0}^{m} \mathbb{P}(S_m = k)X^k = q^{n-m}\sum_{k=0}^{m-1} \mathbb{P}(S_{m-1} = k)X^k + (1-q^{n-m})X\sum_{k=1}^{m} \mathbb{P}(S_{m-1} = k-1)X^{k-1}$$

$$\sum_{k=0}^{m} \mathbb{P}(S_m = k)X^k = q^{n-m}F_{m-1}(X) + (1-q^{n-m})X\sum_{k=0}^{m-1} \mathbb{P}(S_{m-1} = k)X^k$$

$$\sum_{k=0}^{m} \mathbb{P}(S_m = k)X^k = q^{n-m}F_{m-1}(X) + (1-q^{n-m})XF_{m-1}(X)$$

$$F_m(X) = \left(q^{n-m} + (1-q^{n-m})X\right)F_{m-1}(X)$$

Par hypothèse d'induction, on obtient :

$$F_m(X) = \left(q^{n-m} + (1-q^{n-m})X\right)\prod_{k=1}^{m-1} \left(q^{n-k} + (1-q^{n-k})X\right)$$

$$= \prod_{k=1}^{m} \left(q^{n-k} + (1-q^{n-k})X\right)$$

Ce qui prouve le résultat. ■

Le théorème suivant (voir [Bre97]) va nous permettre de déterminer $\mathbb{E}(S_n)$, l'espérance du nombre de sommets sélectionnés par ListLeft une fois sont exécution terminée :

Théorème 21 *Soit S une variable aléatoire discrète non négative et $F(X) = \sum_{k\geq 0} \mathbb{P}(S = k)X^k$ sa série génératrice. Alors,*

$$\begin{aligned} \mathbb{E}(S) &= F'(1) \\ Var(S) &= F''(1) + F'(1) - (F'(1))^2 \end{aligned}$$

En appliquant le théorème 21 au lemme 20, nous obtenons $\mathbb{E}(LL(G_{n,p}))$ et $Var(LL(G_{n,p}))$. Plus précisément,

$$F_n(X) = \prod_{k=1}^{n} q^{n-k} + (1 - q^{n-k})X$$

$$\Rightarrow \log(F_n(X)) = \sum_{k=1}^{n} \log\left(q^{n-k} + (1 - q^{n-k})X\right)$$

$$\Rightarrow (\log(F_n(X)))' = \left(\sum_{k=1}^{n} \log\left(q^{n-k} + (1 - q^{n-k})X\right)\right)'$$

$$\frac{F_n'(X)}{F_n(X)} = \sum_{k=1}^{n} \frac{1 - q^{n-k}}{q^{n-k} + (1 - q^{n-k})X} \tag{3.4}$$

Or, par définition, $F_n(X) = \sum_{k=0}^{n} \mathbb{P}(S_n = k)X^k$, donc $F_n(1) = \sum_{k=0}^{n} \mathbb{P}(S_n = k) = 1$. D'où

$$F_n'(1) = \sum_{k=1}^{n} \frac{1 - q^{n-k}}{q^{n-k} + (1 - q^{n-k})}$$

$$F_n'(1) = \sum_{k=1}^{n} 1 - q^{n-k}$$

$$F_n'(1) = n - \sum_{k=0}^{n-1} q^k$$

$$F_n'(1) = n - \frac{1 - q^n}{1 - q}$$

Par le théorème 21 et le lemme 20 on obtient que $\mathbb{E}(LL(G_{n,p})) = n - \frac{1-q^n}{1-q}$ et en dérivant une fois de plus l'équation 3.4, nous pouvons obtenir $Var(LL(G_{n,p}))$:

$$\frac{F_n''(X)F_n(X) - (F_n'(X))^2}{F_n(X)^2} = -\sum_{k=1}^{n} \frac{(1 - q^{n-k})^2}{(q^{n-k} + (1 - q^{n-k})X)^2}$$

Et comme $F_n(1) = 1$ on obtient :

$$F_n''(1) - (F_n'(1))^2 = -\sum_{k=1}^{n} (1 - q^{n-k})^2$$

$$\text{D'où } F_n''(1) + F_n'(1) - (F_n'(1))^2 = n - \frac{1 - q^n}{1 - q} - \sum_{k=1}^{n} \left(1 - q^{n-k}\right)^2$$

$$= n - \frac{1-q^n}{1-q} - \sum_{k=1}^{n} 1 - 2q^{n-k} + q^{2(n-k)}$$

$$= -\frac{1-q^n}{1-q} + 2\sum_{k=1}^{n} q^{n-k} - \sum_{k=1}^{n} q^{2(n-k)}$$

$$= \frac{1-q^n}{1-q} - \sum_{k=0}^{n-1} q^{2k}$$

$$= \frac{1-q^n}{1-q} - \frac{1-q^{2n}}{1-q^2}$$

Et comme le théorème 21 nous indique que $F_n''(1) + F_n'(1) - (F_n'(1))^2 = Var(LL(G_{n,p}))$, on obtient

$$Var(LL(G_{n,p})) = \frac{1-q^n}{1-q} - \frac{1-q^{2n}}{1-q^2}$$

3.3.2 Espérance et variance de ListRight

Soit S_m le nombre de sommets sélectionnés par ListRight lorsque m sommets ont été parcourus. Comme dans la preuve précédente, on voit que la taille de la solution retournée par ListRight est égale à S_n. On note F_m la série génératrice de S_m, c'est-à-dire le polynôme définit par $F_m(X) = \sum_{k=0}^{m} \mathbb{P}(S_m = k)X^k$.

Lemme 22

$$\forall m \geq 1, F_m(X) = \sum_{k=0}^{m-1} \binom{m-1}{k} X^{m-1-k} q^k \prod_{l=0}^{k-1} (q^l - X) \tag{3.5}$$

avec, par convention, $\prod_{l=0}^{-1}(q^l - X) = 1$.

Preuve. Nous prouvons le lemme par induction sur m. Le premier sommet parcouru par ListRight n'est jamais sélectionné car il ne possède pas de voisin à droite dans la liste, donc $\mathbb{P}(S_1 = 0) = 1$ et $\mathbb{P}(S_1 = 1) = 0$, d'où $F_1(X) = 1$. Pour $m = 1$, la formule 3.5 du lemme 22 vaut $\binom{0}{0} X^0 q^0 \prod_{l=0}^{-1}(q^l - X) = 1$ ce qui vérifie la formule pour $m = 1$.

Pour $m \geq 2$, nous pouvons écrire l'égalité suivante pour tout $k \geq 0$:

$$\mathbb{P}(S_m = k) = \mathbb{P}(S_m = k | S_{m-1} = k)\mathbb{P}(S_{m-1} = k) + \mathbb{P}(S_m = k | S_{m-1} = k-1)\mathbb{P}(S_{m-1} = k-1) \tag{3.6}$$

$\mathbb{P}(S_m = k | S_{m-1} = k)$ est la probabilité pour le sommet m de ne pas être sélectionné, sachant que k sommets ont déjà été sélectionnés auparavant. Ce sommet n'est pas sélectionné uniquement si tous ses voisins droit font partie des k sommets déjà sélectionnés. Cet événement arrive avec probabilité q^{m-1-k} [†]. Avec les mêmes arguments, $\mathbb{P}(S_m = k | S_{m-1} = k-1) = 1 - q^{(m-1)-(k-1)}$. Ainsi, l'égalité 3.6 nous donne

[†]En effet, cet événement revient à dire que le sommet m ne possède pas de voisin parmi les sommets non sélectionnés qui se trouvent à sa droite dans la liste. Comme il y a exactement $m-1-k$ sommets non sélectionnés à droite dans la liste et que q est la probabilité que deux sommets ne soient pas voisins, on obtient que cet événement arrive avec probabilité q^{m-1-k}.

$$\mathbb{P}(S_m = k) = q^{m-1-k}\mathbb{P}(S_{m-1} = k) + (1 - q^{(m-1)-(k-1)})\mathbb{P}(S_{m-1} = k-1)$$

$$\begin{aligned}\Rightarrow \sum_{k=0}^{m}\mathbb{P}(S_m = k)X^k &= q^{m-1}\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k)q^{-k}X^k \\ &+ X\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k-1)X^{k-1} - q^{m-1}\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k-1)q^{-k+1}X^k\end{aligned}$$

et comme $\mathbb{P}(S_{m-1} = -1) = 0$ et $\mathbb{P}(S_{m-1} = m) = 0$, on a :

$$\Rightarrow F_m(X) = q^{m-1}F_{m-1}\left(\frac{X}{q}\right) + XF_{m-1}(X) - q^{m-1}\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k-1)q^{-k+1}X^k$$

or,

$$q^{m-1}\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k-1)q^{-k+1}X^k$$

$$= q^{m-1}X\sum_{k=0}^{m}\mathbb{P}(S_{m-1} = k-1)\frac{X^{k-1}}{q^{k-1}}$$

$$= q^{m-1}X\sum_{k=-1}^{m-1}\mathbb{P}(S_{m-1} = k)\left(\frac{X}{q}\right)^k$$

et comme $\mathbb{P}(S_{m-1} = -1) = 0$

$$= q^{m-1}X\sum_{k=0}^{m-1}\mathbb{P}(S_{m-1} = k)\left(\frac{X}{q}\right)^k = q^{m-1}XF_{m-1}(\frac{X}{q})$$

et on obtient :

$$F_m(X) = q^{m-1}(1-X)F_{m-1}(\frac{X}{q}) + XF_{m-1}(X)$$

En supposant notre proposition vraie pour $m-1$, nous obtenons

$$\begin{aligned}F_m(X) &= q^{m-1}(1-X)\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^{-m+2+k}q^k\prod_{l=0}^{k-1}(q^l - \frac{X}{q}) \\ &+ X\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^k\prod_{l=0}^{k-1}(q^l - X)\end{aligned}$$

$$= (1-X)\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^{2k+1}\prod_{l=0}^{k-1}(q^l - \frac{X}{q}) + \sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-1-k}q^k\prod_{l=0}^{k-1}(q^l - X)$$

$$= (1-X)\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^{k+1}\prod_{l=0}^{k-1}(q^{l+1}-X)+\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

$$= (1-X)\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^{k+1}\prod_{l=1}^{k}(q^{l}-X)+\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

$$= \sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-2-k}q^{k+1}\prod_{l=0}^{k}(q^{l}-X)+\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

$$= \sum_{k=1}^{m-1}\binom{m-2}{k-1}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)+\sum_{k=0}^{m-2}\binom{m-2}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

et comme $\binom{m}{-1}=0$ et $\binom{m}{m+1}=0$, on a

$$F_m(X) = \sum_{k=0}^{m-1}\binom{m-2}{k-1}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)+\sum_{k=0}^{m-1}\binom{m-2}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

de plus, $\binom{m-2}{k-1}+\binom{m-2}{k}=\binom{m-1}{k}$, ce qui nous donne

$$F_m(X) = \sum_{k=0}^{m-1}\binom{m-1}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)$$

ce qui prouve le résultat.

En dérivant $F_m(X)$ et en prenant sa valeur pour $X=1$ nous obtenons la deuxième partie du théorème :

$$(F_m(X))' = \sum_{k=0}^{m-1}\left(\binom{m-1}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)\right)'$$

$$(F_m(X))' = \left(X^{m-1}+(m-1)X^{m-2}q(1-X)\right)' + \sum_{k=2}^{m-1}\left(\binom{m-1}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)\right)'$$

$$(F_m(X))' = (m-1)X^{m-2}+(m-2)(m-1)X^{m-3}q(1-X)-(m-1)X^{m-2}q + \sum_{k=2}^{m-1}\left(\binom{m-1}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)\right)'$$

$$(F_m(X))' = p(m-1)X^{m-2}+q(m-2)(m-1)(1-X)X^{m-3}+\sum_{k=2}^{m-1}\left(\binom{m-1}{k}X^{m-1-k}q^{k}\prod_{l=0}^{k-1}(q^{l}-X)\right)'$$

$$(F_m(X))' = p(m-1)X^{m-2}+q(m-2)(m-1)(1-X)X^{m-3}+\sum_{k=2}^{m-1}\left(\binom{m-1}{k}X^{m-1-k}q^{k}(1-X)\prod_{l=1}^{k-1}(q^{l}-X)\right)'$$

Comme $(a(x)b(x)c(x))' = a'(x)b(x)c(x) + a(x)b'(x)c(x) + a(x)b(x)c'(x)$ et que $(q^l - X)' = -1$, on a :

$$\begin{aligned}(F_m(X))' &= p(m-1)X^{m-2} + q(m-2)(m-1)(1-X)X^{m-3} \\ &+ \sum_{k=2}^{m-1} \binom{m-1}{k} q^k \left((m-1-k)X^{m-2-k}(1-X) \prod_{l=1}^{k-1}(q^l - X) \right. \\ &\left. -X^{m-1-k} \prod_{l=1}^{k-1}(q^l - X) - X^{m-1-k}(1-X) \sum_{l=1}^{k-1} \left(\prod_{j=1}^{l-1} (q^j - X) \prod_{j=l+1}^{k-1} (q^j - X) \right) \right)\end{aligned}$$

Et lorsqu'on évalue $(F_m(X))'$ en 1 on obtient :

$$\begin{aligned}&p(m-1)1^{m-2} + q(m-2)(m-1)(1-1)1^{m-3} + \sum_{k=2}^{m-1} \binom{m-1}{k} q^k \left((m-1-k)1^{m-2-k}(1-1) \prod_{l=1}^{k-1}(q^l - 1) \right. \\ &\left. -1^{m-1-k} \prod_{l=1}^{k-1}(q^l - 1) - 1^{m-1-k}(1-1) \sum_{l=1}^{k-1} \left(\prod_{j=1}^{l-1} (q^j - 1) \prod_{j=l+1}^{k-1} (q^j - 1) \right) \right) \\ &= p(m-1) + \sum_{k=2}^{m-1} \binom{m-1}{k} q^k \left(- \prod_{l=1}^{k-1} \left(q^l - 1 \right) \right) \\ &= p(m-1) + \sum_{k=2}^{m-1} \binom{m-1}{k} (-q)^k \left(\prod_{l=1}^{k-1} \left(1 - q^l \right) \right)\end{aligned}$$

Pour obtenir la variance, nous devons calculer $(F_m(X))''$. L'expression de $(F_m(X))'$ étant relativement longue et n'apportant rien de nouveau en terme d'idées, nous n'en ferons pas la démonstration ici. Cependant, $(F_m(X))''$ se calcule avec les mêmes méthodes et arguments que pour le calcul de $(F_m(X))'$. ■

3.4 Bilan

Dans ce chapitre, nous avons montré que Sorted ListRight possède un comportement online, et qu'il retourne la même solution que l'algorithme online de Demange et Paschos lorsqu'ils considèrent les sommets dans le même ordre. Ainsi, tous les résultats obtenus pour l'un des algorithmes est aussi valable pour l'autre, ce qui nous permet d'affirmer, par exemple, que lorsque l'ordre de révélation des sommets s'effectue dans l'ordre croissant des degrés, l'algorithme online va retourner une solution au plus $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}-$approchée. Au contraire, l'algorithme Sorted ListLeft n'a pas besoin de mémoire puisque son choix est déterminé par les relation du sommet courant avec les sommets non encore parcourus, ce qui en fait un bon candidat pour traiter de très grandes instances. Nous avons ensuite relâché la contrainte de tri des listes car elle ne convenait pas, d'après nous, à la philosophie des algorithmes de listes, qui doivent traiter les instances le plus rapidement possible. Le retrait de cette contrainte a dégradé le pire cas des deux algorithmes et nous avons évalué leurs performances en moyenne, sous réserve d'équiprobabilité des listes. Nous avons exhibé

une classe de graphes pour laquelle l'algorithme ListLeft est $\left(\frac{n+1}{2} - \frac{1}{n}\right)$ −approché en moyenne, avec n le nombre de sommets du graphe et que, pour tout graphe, l'algorithme ListRight est au plus $\left(2 - \frac{2}{n}\right)$ −approché, ce qui en fait un bon candidat pour concurrencer les meilleurs algorithmes. Pour montrer ce résultat, nous avons du utiliser le fait que l'algorithme ListRight retourne la même solution que *LRglouton* (un algorithme glouton) lorsqu'ils considèrent les sommets dans le même ordre. Ainsi, les trois algorithmes ListRight, online et *LRglouton* se comportent de façon identique et l'étude de ce phénomène est une perspective très intéressante car cela permet d'obtenir des résultats relativement facilement. Nous avons ensuite poursuivi la comparaison des deux algorithmes de liste sur les graphes aléatoire de Erdös-Renyi. Nous avons donné un moyen de calculer l'espérance de la taille des solutions retournées pour chacun des deux algorithmes et nous avons montré que la différence entre leurs performances moyennes est significative. Tous ces arguments montrent que l'utilisation de ListRight est donc fortement préférable à celle de ListLeft.

Chapitre 4

Calculs analytiques de l'espérance de la taille de vertex cover construits par divers algorithmes sur les chemins

Dans les chapitres précédents, nous avons mené une étude comparative des algorithmes ListLeft et ListRight. Nous avons montré que ListRight est toujours meilleur du point de vue de la qualité de la solution construite et nous avons fourni une majoration de l'espérance de son rapport d'approximation, démontrant ainsi que pour tout graphe, les performances moyennes de ListRight sont bonnes, contrairement à ce que laissait supposer son rapport d'approximation en pire cas. Cependant, cette majoration est très proche de 2 et de très nombreux graphes n'admettent pas de solution dont le rapport d'approximation soit supérieur à 2 (comme les chemins, les grilles ou les hypercubes par exemple). On peut alors se demander quel est le comportement exact de ListRight lorsque l'on considère une classe de graphes précise qui n'admet pas de solution dont le rapport d'approximation en pire cas est supérieur à 2. C'est l'objet de ce chapitre. Afin de poursuivre notre démarche comparative, nous étendons ce travail à tous les algorithmes utilisés jusqu'ici tout en enrichissant notre panel d'algorithmes par les algorithmes Greedy Independent Cover et Depth First Search.

Dans ce chapitre, nous menons une étude approfondie du comportement exact des différents algorithmes sur une classe de graphes simple mais non triviale : les chemins*. Pour cela, nous donnons une formule close et exacte de l'espérance de la taille des solutions retournées pour chaque algorithme et nous calculons la limite de leur rapport d'approximation lorsque la taille du chemin tend vers l'infini. Nous complétons ensuite notre étude par une démarche expérimentale, ce qui nous permettra de déterminer la fonction de répartition des différentes solutions obtenues lorsqu'on exécute un grand nombre de fois le même algorithme sur un même chemin, ce qui revient à donner la probabilité qu'un algorithme retourne une solution de taille k lorsqu'il est appliqué sur un chemin.

Dans la suite de ce chapitre, on notera P_n un chemin de taille n et ses sommets seront numéroté de u_1 à u_n. A diverses reprises, nous utiliserons l'égalité suivante (voir [AS64]) : $e^{-2} = \sum_{k=0}^{\infty} \frac{(-2)^k}{k!}$.

*Un chemin de taille n est un graphe connexe à n sommets et $n-1$ arêtes et dont tous les sommets sont de degré 2 sauf deux qui sont de degré 1.

4.1 Deux algorithmes supplémentaires : Greedy Independent Cover et Depth First Search

Dans cette section, Nous présentons deux nouveaux algorithmes pour le problème du vertex cover.

L'algorithme Depth First Search, présenté dans [Sav82], exploite le fait que les sommets internes d'un arbre de parcours en profondeur forment une couverture de sommets dont la taille est 2−approchée en pire cas. Cette stratégie retourne une couverture connexe ce qui rend cet algorithme pratiquement déterministe[†] et peu efficace sur les chemins bien qu'il possède un très bon rapport d'approximation.

Algorithme 5 : Algorithme DFS (Depth First Search)

Données : Un graphe G
Sortie : Une couverture des sommets de G
Calculer T, un arbre de parcours en profondeur du graphe G.
Déterminer $I(T)$, l'ensemble des nœuds internes de T.
retourner $I(T)$;

Espérance de l'algorithme Depth First Search sur les chemins. L'algorithme va construire un arbre de parcours en profondeur. Si le sommet de départ du parcours est un sommet interne de P_n (c'est-à-dire un sommet de degré 2), alors l'arbre de parcours en profondeur construit possèdera deux feuilles : les deux sommets de degré 1 situés aux extrémités du chemin. Un arbre de parcours en profondeur étant un graphe orienté, une feuille est un sommet qui ne possède pas de fils. Si le sommet de début du parcours est un sommet de degré 1, alors l'arbre construit n'aura qu'une seule feuille, l'autre sommet de degré 1, car le premier sommet possède un fils et n'est pas une feuille par définition.

Le choix du sommet de départ étant équiprobable (voir définition 16), on peut en déduire l'espérance de la taille de la solution retournée lorsqu'on applique Depth First Search sur P_n. En effet, si le sommet de départ est l'un des deux sommets de degré 1, alors l'algorithme va retourner une solution de taille $n-1$ et dans les $n-2$ autres cas, la solution retournée sera de taille $n-2$.

$$\text{d'où } \mathbb{E}(DFS(P_n)) = \frac{2}{n}(n-1) + \frac{n-2}{n}(n-2) = n-2+\frac{2}{n}$$

Nous allons maintenant déterminer la limite de l'espérance du rapport d'approximation. Pour cela, nous avons besoin du lemme suivant :

Lemme 23 *Soit $opt(P_n)$ la taille d'un vertex cover optimal sur P_n et $A(P_n)$ la taille de la solution retournée par un algorithme A appliqué sur P_n. On a :*

$$\mathbb{E}\left(\frac{A(P_n)}{opt(P_n)}\right) = \frac{\mathbb{E}\left(A(P_n)\right)}{opt(P_n)}$$

[†]L'algorithme Depth First Search est pratiquement déterministe sur les chemins dans le sens où toutes les solutions pouvant être retournées sont identiques à un ou deux sommets près.

Preuve. Lorsque n est donné, $opt(P_n) = \lfloor \frac{n}{2} \rfloor$ est constant. Soit $\mathbb{E}(A(P_n)) = \sum_{i=0}^{n} p_i \cdot x_i$ l'espérance de la taille des solutions retournées par l'algorithme A sur P_n, avec p_i la probabilité que l'algorithme A retourne une solution de taille x_i. En utilisant la même notation, on a $\mathbb{E}\left(\frac{A(P_n)}{opt(P_n)}\right) = \sum_{i=0}^{n} p_i \cdot \frac{x_i}{opt(P_n)}$. Comme $opt(P_n)$ est constant, on a directement $\mathbb{E}\left(\frac{A(P_n)}{opt(P_n)}\right) = \frac{1}{opt(n)} \sum_{i=0}^{n} p_i \cdot x_i = \frac{\mathbb{E}(A(P_n))}{opt(P_n)}$. ■

Par le lemme 23, sachant que $\mathbb{E}(DFS(P_n)) = n - 2 + \frac{2}{n}$ et que $opt(P_n) = \lfloor \frac{n}{2} \rfloor$, on obtient trivialement que la limite du rapport d'approximation de Depth First Search tend vers 2 lorsque la taille du chemin tend vers l'infini.

L'algorithme *GIC* est basé sur l'idée que sélectionner un sommet de degré minimum est un choix pouvant s'avérer peu judicieux. Par exemple, sélectionner une feuille d'un arbre est un mauvais choix, car si on sélectionne son voisin, on est sûr de couvrir l'arête les reliant, et on peut (potentiellement) en couvrir d'autres. Ainsi, sélectionner le voisin d'une feuille est un choix optimal. Cet algorithme va donc choisir un sommet de degré minimum (sans le placer dans la solution) et va ajouter tout son voisinage dans la solution.

Algorithme 6 : Algorithme GIC (Greedy Independent Cover)

Data : Un graphe $G = (V, E)$
Result : Une couverture des sommets de G
$C \leftarrow \emptyset$;
while $E \neq \emptyset$ **do**
  Choisir un sommet u de degré minimum;
  $C \leftarrow C \cup N(u)$;
  $V \leftarrow V - (N(u) \cup \{u\})$;
end
return C;

Cet algorithme possède un mauvais rapport d'approximation en pire cas. En effet, si on l'applique aux graphes présentés dans [AI07] pour piéger Sorted ListLeft, Greedy Independent Cover va retourner une solution $\frac{\sqrt{\Delta}}{2}-$approchée.

Espérance de Greedy Independent Cover sur les chemins. Cet algorithme possède un comportement optimal sur les chemins. Un chemin possède deux sommets de degré 1 (situés sur les bords du chemin) et tous les autres sommets sont de degré 2. *GIC* va donc considérer un des deux sommets du bord, sélectionner son unique voisin et retirer ces deux sommets du graphe. On va donc se retrouver avec un chemin de taille P_{n-2} en ayant sélectionné un seul sommet. Ainsi, étape par étape, l'algorithme va réduire le chemin, jusqu'a ce qu'on obtienne un chemin P_1 ou P_0 (qui ne possèdent pas d'arêtes). Ce comportement est optimal. En effet, il existe une exécution de l'algorithme Edge Deletion qui va sélectionner exactement deux fois plus de sommets que *GIC* : à chaque fois que *GIC* considère un sommet u et sélectionne son voisin v, Edge Deletion peut sélectionner les deux sommets de l'arête uv, ajoutant ainsi deux fois plus de sommets que *GIC*. Comme Edge Deletion est 2−approché, cela signifie que *GIC* retourne une solution optimale. Plus généralement et avec les mêmes arguments, on remarque que *GIC* est optimal sur tous les arbres.

4.2 Expression de l'espérance de ListLeft sur les chemins

L'évaluation en moyenne de ListLeft se fait très facilement. Dans une communication personnelle, Eric Angel a montré que pour un graphe G de taille n, $\mathbb{E}(\text{ListLeft}(G)) = n - \sum_{u \in V} \frac{1}{d(u)+1}$. Pour s'en convaincre, il suffit de remarquer qu'un sommet u n'est pas sélectionné uniquement si tout son voisinage se trouve à sa gauche dans la liste. En considérant toutes les listes respectant cette condition, on obtient que la probabilité d'un tel évènement est $\frac{1}{d(u)+1}$. La probabilité de sélectionner un sommet s'en déduit directement : $1 - \frac{1}{d(u)+1}$ ainsi que la formule annoncée.

Dans un chemin de taille n, on a exactement $n-2$ sommets de degré 2 et deux sommets de degré 1. Lorsqu'on applique cette formule aux chemins, on obtient

$$\mathbb{E}(\text{ListLeft}(P_n)) = n - \frac{1}{3}(n-2) - \frac{1}{2}2 = \frac{2n-1}{3}$$

La limite de l'espérance du rapport d'approximation s'obtient facilement car

$$\frac{2}{3}\frac{2n-1}{n-1} = \frac{\frac{2n-1}{3}}{\frac{n-1}{2}} \geq \frac{\frac{2n-1}{3}}{\left\lfloor \frac{n}{2} \right\rfloor} \geq \frac{\frac{2n-1}{3}}{\frac{n}{2}} = \frac{2}{3}\frac{2n-1}{n}.$$

Ainsi, lorsque la taille du chemin tend vers l'infini, l'espérance du rapport d'approximation de ListLeft tend vers $\frac{4}{3}$.

4.3 Expression de l'espérance de ListRight sur les chemins

Nous avons déterminé l'espérance exacte des algorithmes Depth First Search, Greedy Independent Cover et ListLeft de façon très simple. Le calcul de l'espérance des algorithmes ListRight, Edge Deletion et Maximum Degree Greedy ne s'effectue pas aussi directement. Nous allons tout d'abord déterminer une expression récursive de l'espérance de la taille de la solution retournée, puis nous allons déterminer une formule alternative close en utilisant la méthode suivante :

Méthode de résolution. Soit u_n une suite définie récursivement, et $F(X) = \sum_{n \geq 0} u_n X^n$ sa fonction génératrice. Nous allons exprimer $F(X)$ en fonction de $F'(X)$ de manière à obtenir une équation différentielle. En résolvant cette équation différentielle, nous obtenons une formulation alternative de $F(X)$ et par un jeu de réécriture impliquant certaines fonctions génératrices ordinaires usuelles, on obtient $F(X) = \sum_{n \geq 0} v_n X^n$, avec v_n une suite définie de manière non récursive. Par définition, on a $u_n = v_n$, obtenant ainsi une expression non récursive de u_n.

Cette méthode permet à la fois de trouver et de prouver la formulation close de l'espérance. Dans un souci de lisibilité, le déroulement de cette méthode ne sera donné qu'en annexe à la fin du du chapitre et seulement pour l'algorithme ListRight car elle est identique pour chacun des trois algorithmes. Nous ne donnerons ici qu'une preuve simple qui vérifie uniquement le résultat pour chaque algorithme.

La probabilité qu'un sommet u de P_n soit révélé en premier, c'est-à-dire qu'il se trouve à l'extrémité droite de la liste, est $\frac{1}{n}$ (voir définition 16). On a :

Théorème 24 *L'espérance du nombre de sommets retournés par* ListRight *sur* P_n *vaut :*

$$\mathbb{E}(LR(P_n)) = \frac{n-1}{2} + \frac{n+1}{2} \cdot \sum_{k=0}^{n} \frac{(-2)^k}{k!} + \sum_{k=0}^{n-1} \frac{(-2)^k}{k!}$$

Preuve. Le lemme 25 montre que $\mathbb{E}(LR(P_n)) = \frac{2}{n}\left(n-1+\sum_{k=2}^{n-2}\mathbb{E}(LR(P_k))\right)$ et le lemme 26 vérifie que $\frac{2}{n}\left(n-1+\sum_{k=2}^{n-2}\mathbb{E}(LR(P_k))\right) = \frac{n-1}{2} + \frac{n+1}{2}\cdot\sum_{k=0}^{n}\frac{(-2)^k}{k!} + \sum_{k=0}^{n-1}\frac{(-2)^k}{k!}$. Une deuxième preuve du théorème 24 est donnée ensuite dans l'annexe du présent chapitre. Cette deuxième preuve montre la démarche utilisée pour trouver le résultat. ■

Lemme 25

$$\mathbb{E}(LR(P_0)) = \mathbb{E}(LR(P_1)) = 0, \mathbb{E}(LR(P_n)) = \frac{2}{n}\left(n-1+\sum_{k=0}^{n-2}\mathbb{E}(LR(P_k))\right)$$

Preuve. Dans cette preuve, nous considérons la version gloutonne de l'algorithme. ListRight ne sélectionne aucun sommet si la taille du chemin est plus petite que 2. En effet, ListRight ne sélectionne un sommet que s'il possède un voisin droit, ce qui ne peut être le cas que si le graphe possède au moins une arête et donc au moins 2 sommets. Ainsi, $\mathbb{E}(LR(P_0)) = \mathbb{E}(LR(P_1)) = 0$.

Soit u_k le premier sommet observé par ListRight. ListRight va sélectionner le voisinage de u_k, c'est-à-dire le sommet u_2 si $k = 1$, le sommet u_{n-1} si $k = n$, les sommets u_{k-1} et u_{k+1} dans tous les autres cas.

Après cette sélection, nous obtenons deux chemins (éventuellement vides) résultants de la suppression de u_k et de $N(u_k)$: RP_1 et RP_2. Plusieurs cas sont possibles :

- Si $k = 1$ alors ListRight sélectionne un sommet et $RP_1 = P_0$ et $RP_2 = P_{n-2}$.
- Si $k = n$ alors ListRight sélectionne un sommet et $RP_1 = P_{n-2}$ et $RP_2 = P_0$.
- Si $k = 2$ alors ListRight sélectionne deux sommets et $RP_1 = P_0$ et $RP_2 = P_{n-3}$.
- Si $k = n-1$ alors ListRight sélectionne deux sommets et $RP_1 = P_{n-3}$ et $RP_2 = P_0$.
- Dans tous les autres cas, ListRight sélectionne deux sommets et RP_1 est composé des sommets u_1 à u_{k-2}, et RP_2 est composé par les sommets u_{k+2} à u_n. Ainsi, comme la taille du chemin RP_1 est $k-2$ et la taille de RP_2 est $n-(k+1)$, nous obtenons que $RP_1 = P_{k-2}$ et $RP_2 = P_{n-(k+1)}$.

Comme chaque sommet à une probabilité de $\frac{1}{n}$ d'être le premier sommet de la liste, nous pouvons en déduire une expression de l'espérance de la taille de la solution retournée par ListRight sur P_n :

$$\mathbb{E}(LR(P_n)) = \underbrace{2\cdot\frac{1}{n}\cdot\left(1+\mathbb{E}\left(LR\left(P_0\right)\right)+\mathbb{E}\left(LR\left(P_{n-2}\right)\right)\right)}_{pour\ k=1\ et\ k=n} + \underbrace{\sum_{k=2}^{n-1}\frac{1}{n}\cdot\left(2+\mathbb{E}\left(LR\left(P_{k-2}\right)+\mathbb{E}\left(LR\left(P_{n-(k+1)}\right)\right)\right)\right)}_{pour\ k=2\ jusque\ n-1}$$

En décomposant la somme, et en notant que $\mathbb{E}\left(LR\left(P_0\right)\right) = 0$, on obtient :

$$\mathbb{E}(LR(P_n)) = \frac{2}{n}\cdot(1+\mathbb{E}(LR(P_{n-2}))) + \sum_{k=2}^{n-1}\frac{1}{n}\cdot 2 + \sum_{k=2}^{n-1}\frac{1}{n}\cdot\mathbb{E}(LR(P_{k-2}) + \sum_{k=2}^{n-1}\frac{1}{n}\cdot\mathbb{E}(LR(P_{n-(k+1)})))$$

On s'aperçoit en réécrivant les termes des deux dernières sommes qu'elles sont égales :

$$\sum_{k=2}^{n-1}\frac{1}{n}\cdot\mathbb{E}(LR(P_{k-2}) = \frac{1}{n}\left(\mathbb{E}(LR(P_0)+\mathbb{E}(LR(P_1)+\cdots+\mathbb{E}(LR(P_{n-3}))\right)$$

$$\sum_{k=2}^{n-1} \frac{1}{n} \cdot \mathbb{E}(LR(P_{n-(k+1)}))) = \frac{1}{n} \left(\mathbb{E}(LR(P_{n-3}) + \mathbb{E}(LR(P_{n-2}) + \cdots + \mathbb{E}(LR(P_0))\right)$$

Nous pouvons reformuler notre expression de manière plus simple :

$$\mathbb{E}(LR(P_n)) = \frac{2}{n} \cdot (1 + \mathbb{E}(LR(P_{n-2}))) + \sum_{k=2}^{n-1} \frac{1}{n} \cdot 2 + 2 \cdot \sum_{k=2}^{n-1} \frac{1}{n} \cdot \mathbb{E}(LR(P_{k-2}))$$

$$\mathbb{E}(LR(P_n)) = \frac{2}{n} \cdot (1 + \mathbb{E}(LR(P_{n-2}))) + 2 \cdot \sum_{k=2}^{n-1} \frac{1}{n} \cdot (1 + \mathbb{E}(LR(P_{k-2})))$$

$$\mathbb{E}(LR(P_n)) = 2 \cdot \sum_{k=2}^{n} \frac{1}{n} \cdot (1 + \mathbb{E}(LR(P_{k-2})))$$

$$\mathbb{E}(LR(P_n)) = 2 \cdot \sum_{k=0}^{n-2} \frac{1}{n} \cdot (1 + \mathbb{E}(LR(P_k)))$$

$$\mathbb{E}(LR(P_n)) = \frac{2}{n} \cdot \sum_{k=0}^{n-2} 1 + \mathbb{E}(LR(P_k))$$

et nous obtenons, finalement :

$$\mathbb{E}(LR(P_n)) = \frac{2}{n} \cdot \left(n - 1 + \sum_{k=0}^{n-2} \mathbb{E}(LR(P_k))\right)$$

■

Nous avons désormais une expression récursive de l'espérance de ListRight sur les chemins. Cependant, cette expression est difficilement utilisable car elle est compliquée à manipuler et à analyser. En utilisant les séries génératrices, nous avons réussi à « dérécursiver » cette expression. Pour plus de détails sur les séries génératrices, le lecteur est invité à consulter cet excellent ouvrage [Wil06].

Lemme 26 *Pour $n \geq 2$, on a :*

$$\mathbb{E}(LR(P_n)) = \frac{2}{n} \left(n - 1 + \sum_{k=0}^{n-2} u_k\right) = \frac{n-1}{2} + \frac{n+1}{2} \cdot \sum_{k=0}^{n} \frac{(-2)^k}{k!} + \sum_{k=0}^{n-1} \frac{(-2)^k}{k!}$$

Preuve. Le lemme 25 nous donne une expression récursive de $\mathbb{E}(\text{ListRight}(P_n))$. Nous avons utilisé les séries génératrices pour trouver la formule close du théorème 24. Dans cette preuve, nous donnons des arguments simples permettant de vérifier la validité du théorème 24. Pour cela, on note : $u_n = \mathbb{E}(\text{ListRight}(P_n))$ (avec $u_0 = 0, u_1 = 0$). Nous devons juste prouver que pour tout $n \geq 2$:

$$u_n = \frac{2}{n} \left(n - 1 + \sum_{k=2}^{n-2} u_k\right) = \frac{n-1}{2} + \frac{n+1}{2} \cdot \sum_{k=0}^{n} \frac{(-2)^k}{k!} + \sum_{k=0}^{n-1} \frac{(-2)^k}{k!} \tag{4.1}$$

Dans un premier temps, nous réarrangeons l'expression de u_n comme suit :

$$u_n = \frac{2}{n}\left(n-2+\sum_{k=2}^{n-3} u_k + 1 + u_{n-2}\right) = \frac{2}{n}\left(\frac{n-1}{2}u_{n-1} + 1 + u_{n-2}\right)$$

on obtient

$$\frac{n}{n-1}u_n = 2+\frac{2}{n-1}\sum_{k=2}^{n-2} u_k = 2+\frac{2}{n-1}\sum_{k=2}^{n-3} u_k + \frac{2}{n-1}u_{n-2} = \frac{2(n-2)}{n-1} + \frac{2}{n-1}\sum_{k=2}^{n-3} u_k + \frac{2}{n-1} + \frac{2}{n-1}u_{n-2}$$

Comme $u_{n-1} = \frac{2(n-2)}{n-1} + \frac{2}{n-1}\sum_{k=2}^{n-3} u_k$, on a : $\frac{n}{n-1}u_n = u_{n-1} + \frac{2}{n-1}(u_{n-2}+1)$ et finalement on obtient :

$$u_n = \frac{2}{n}\left(n-2+\sum_{k=2}^{n-3} u_k + 1 + u_{n-2}\right) = \frac{2}{n}\left(\frac{n-1}{2}u_{n-1} + 1 + u_{n-2}\right) = \frac{1}{n}\left((n-1)u_{n-1} + 2u_{n-2} + 2\right) \tag{4.2}$$

Maintenant nous allons vérifier le résultat du théorème par induction sur n. C'est vrai pour $n = 2$. Nous supposons que le résultat est vrai jusqu'au rang $n-1$ et nous le vérifions pour le rang n. Pour simplifier, on note $b_j = \sum_{k=0}^{j} \frac{(-2)^k}{k!}$ et $a_k = \frac{(-2)^k}{k!}$. Dans (4.2) nous remplaçons u_{n-1} et u_{n-2} par leur expression donnée par l'hypothèse d'induction. Nous obtenons :

$$u_n = \frac{1}{n}((n-1)u_{n-1} + 2u_{n-2} + 2)$$

$$= \frac{1}{n}\left(\frac{(n-1)(n-2)}{2} + \frac{n(n-1)}{2}b_{n-1} + (n-1)b_{n-2} + n - 3 + (n-1)b_{n-2} + 2b_{n-3} + 2\right)$$

$$= \frac{n-1}{2} + \frac{2(n-1)}{n}b_{n-2} + \frac{n-1}{2}b_{n-1} + \frac{2}{n}b_{n-3}$$

Pour obtenir le résultat, nous devons juste prouver que :

$$\frac{2(n-1)}{n}b_{n-2} + \frac{n-1}{2}b_{n-1} + \frac{2}{n}b_{n-3} = \frac{n+1}{2}b_n + b_{n-1}$$

ce qui équivaut à montrer que :

$$\frac{2(n-1)}{n}b_{n-2} + \frac{n-3}{2}b_{n-1} + \frac{2}{n}b_{n-3} = \frac{n+1}{2}b_n$$

En utilisant le fait que $b_{n-k} = b_n - \sum_{l=n-k+1}^{n} a_l$, on obtient

$$\frac{2(n-1)}{n}b_{n-2} + \frac{n-3}{2}b_{n-1} + \frac{2}{n}b_{n-3}$$

$$= \frac{n+1}{2}b_n - \frac{2}{n}(a_{n-2}+a_{n-1}+a_n) - \frac{2(n-1)}{n}(a_{n-1}+a_n) - \frac{n-3}{2}a_n$$

et

$$-\frac{2}{n}(a_{n-2}+a_{n-1}+a_n) - \frac{2(n-1)}{n}(a_{n-1}+a_n) - \frac{n-3}{2}a_n$$

$$= -\frac{2}{n}a_{n-2} - 2a_{n-1} - \frac{n+1}{2}a_n = \frac{(-2)^{n-1}}{n(n-2)!} + \frac{(-2)^n}{(n-1)!} + \frac{(-2)^{n-1}(n+1)}{n!} = 0$$

■

Lemme 27 *La limite de l'espérance du rapport d'approximation de* ListRight *sur P_n lorsque n tend vers l'infini vaut $1 + e^{-2}$.*

Preuve. Le lemme 23 nous indique que $\mathbb{E}\left(\frac{LR(P_n)}{OPT(P_n)}\right) = \frac{\mathbb{E}(LR(P_n))}{OPT(P_n)}$, donc $\lim_{n\to\infty} \mathbb{E}\left(\frac{LR(P_n)}{OPT(P_n)}\right) = \lim_{n\to\infty} \frac{\mathbb{E}(LR(P_n))}{OPT(P_n)} = \lim_{n\to\infty} \frac{\mathbb{E}(LR(P_n))}{\lfloor \frac{n}{2} \rfloor}$.
On a

$$\mathbb{E}(LR(P_n)) = \frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}$$

et

$$\frac{\frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}}{\frac{n}{2}} \leq \frac{\frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}}{\lfloor \frac{n}{2} \rfloor} \leq \frac{\frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}}{\frac{n-1}{2}}$$

On calcule la limite de la partie gauche :

$$\lim_{n\to\infty} \frac{\frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}}{\frac{n}{2}} = \lim_{n\to\infty} \frac{n - 1 + (n+1) b_n + 2 b_{n-1}}{n} = 1 + \lim_{n\to\infty} \frac{n+1}{n} b_n + \lim_{n\to\infty} \frac{2}{n} b_{n-1} = 1 + e^{-2}$$

De la même façon, on calcule la limite de la partie droite :

$$\lim_{n\to\infty} \frac{\frac{n-1}{2} + \frac{n+1}{2} b_n + b_{n-1}}{\frac{n-1}{2}} = \lim_{n\to\infty} \frac{n - 1 + (n+1) b_n + 2 b_{n-1}}{n-1} = 1 + \lim_{n\to\infty} \frac{n+1}{n-1} b_n + \lim_{n\to\infty} \frac{2}{n-1} b_{n-1} = 1 + e^{-2}$$

On en déduit que l'espérance du rapport d'approximation de ListRight tend vers $1 + e^{-2}$ lorsque la taille du chemin tend vers l'infini. ■

Ces résultats montrent que ListRight offre de bons résultats sur les chemins, étant donné que la limite de l'espérance de son rapport d'approximation est relativement éloignée de 2.

4.4 Expression de l'espérance de Edge Deletion sur les chemins

Nous avons vu que Depth First Search, qui est un algorithme 2−approché, retourne des solutions dont la qualité (sur les chemins) est mauvaise. Nous allons maintenant exhiber une formule close de l'espérance de la taille des solutions retournées par Edge Deletion, ce qui nous permettra d'avoir un second regard sur le comportement des algorithmes 2−approchés sur les chemins. Pour déterminer cette formule close, nous avons utilisé la même démarche que pour ListRight et pour faciliter la lecture, nous ne donnerons que la preuve de vérification du résultat.

Théorème 28 *L'espérance du nombre de sommets retournés par* Edge Deletion *sur P_n vaut :*

$$\mathbb{E}(ED(P_n)) = n - n \sum_{p=0}^{n-1} \frac{(-2)^p}{p!} - 2 \sum_{p=0}^{n-2} \frac{(-2)^p}{p!}$$

Lemme 29 *Une expression récursive de l'espérance du nombre de sommets retournés par* Edge Deletion *sur P_n est :*

$$\mathbb{E}(ED(P_0)) = \mathbb{E}(ED(P_1)) = 0$$

$$\mathbb{E}(ED(P_n)) = 2 + \frac{2}{n-1} \sum_{k=2}^{n-2} \mathbb{E}(ED(P_k))$$

Preuve. A chaque étape, Edge Deletion va sélectionner les deux sommets d'une arête du graphe. Si le graphe possède moins de 2 sommets alors il ne contient aucune arête et Edge Deletion ne va sélectionner aucun sommet, d'où $\mathbb{E}(ED(P_0)) = \mathbb{E}(ED(P_1)) = 0$.

Lorsque le nombre de sommets du chemin est supérieur ou égal à 2, Edge Deletion va sélectionner une arête (u_k, u_{k+1}) de façon équiprobable (voir définition 16) parmi les $n-1$ arêtes du chemin.

Après cette sélection, nous obtenons deux chemins (éventuellement vides) résultants de la suppression des sommets u_k et u_{k+1} : RP_1 et RP_2. Nous pouvons distinguer trois cas :

- Edge Deletion sélectionne les sommets de l'arête (u_1, u_2). Dans ce cas, on a $RP_1 = P_0$ et $RP_2 = P_{n-2}$.
- Edge Deletion sélectionne les sommets de l'arête (u_n, u_{n-1}). Dans ce cas, on a $RP_1 = P_{n-2}$ et $RP_2 = P_0$.
- Edge Deletion sélectionne les sommets de l'arête (u_k, u_{k+1}), avec $1 < k < n-1$. Dans ce cas, RP_1 est composé des sommets u_1 à u_{k-1} et RP_2 est composé par les sommets u_{k+2} à u_n. Comme la taille du chemin RP_1 est $k-1$ et la taille de RP_2 est $n-(k+1)$, nous obtenons que $RP_1 = P_{k-1}$ et $RP_2 = P_{n-(k+1)}$.

Nous pouvons en déduire une expression de l'espérance de la taille de la solution retournée par Edge Deletion sur P_n :

$$\mathbb{E}(ED(P_0)) = \mathbb{E}(ED(P_1)) = 0$$

$$\mathbb{E}(ED(P_n)) = \sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \left(2 + \mathbb{E}\left(ED\left(P_{k-1}\right) + \mathbb{E}\left(ED\left(P_{n-(k+1)}\right)\right)\right)\right)$$

En décomposant la somme, on obtient :

$$\mathbb{E}(ED(P_n)) = 2\sum_{k=1}^{n-1} \frac{1}{n-1} + \sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \mathbb{E}(ED(P_{k-1}) + \sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \mathbb{E}(ED(P_{n-(k+1)})))$$

On s'aperçoit, en réécrivant les termes des deux dernières sommes qu'elles sont égales :

$$\sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \mathbb{E}(ED(P_{k-1})) = \frac{1}{n-1} \left(\mathbb{E}(ED(P_0) + \mathbb{E}(ED(P_1) + \mathbb{E}(ED(P_2) + \cdots + \mathbb{E}(ED(P_{n-2}))\right)$$

$$\sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \mathbb{E}(ED(P_{n-(k+1)})) = \frac{1}{n-1} \left(\mathbb{E}(ED(P_{n-2}) + \mathbb{E}(ED(P_{n-3}) + \cdots + \mathbb{E}(ED(P_1) + \mathbb{E}(ED(P_0))\right)$$

Nous pouvons reformuler notre expression de manière plus simple :

$$\mathbb{E}(ED(P_n)) = 2\sum_{k=1}^{n-1} \frac{1}{n-1} + 2 \cdot \sum_{k=1}^{n-1} \frac{1}{n-1} \cdot \mathbb{E}(ED(P_{k-1}))$$

Et nous obtenons finalement

$$\mathbb{E}(ED(P_n)) = 2 + \frac{2}{n-1} \sum_{k=0}^{n-2} \mathbb{E}(ED(P_k))$$

■

Lemme 30 *Pour $n \geq 2$, on a :*

$$\mathbb{E}(ED(P_n)) = 2 + \frac{2}{n-1}\sum_{k=0}^{n-2}\mathbb{E}(ED(P_k)) = n - n\sum_{p=0}^{n-1}\frac{(-2)^p}{p!} - 2\sum_{p=0}^{n-2}\frac{(-2)^p}{p!}$$

Preuve. On note $u_n = \mathbb{E}(ED(P_n))$. On a

$$u_n = \frac{2}{n-1}\left(n-1+\sum_{k=0}^{n-2}u_k\right) = \frac{2}{n-1}\left(n-1+\sum_{k=2}^{n-2}u_k\right) \text{ car } u_0 = u_1 = 0$$

$$\Rightarrow \frac{n-1}{n-2}u_n = \frac{2}{n-2}\left(n-1+\sum_{k=2}^{n-2}u_k\right)$$

$$= \frac{2}{n-2} + \frac{2}{n-2}\left(n-2+\sum_{k=2}^{n-2}u_k\right)$$

$$= \frac{2}{n-2} + \frac{2}{n-2}u_{n-2} + \frac{2}{n-2}\left(n-2+\sum_{k=2}^{n-3}u_k\right)$$

$$= \frac{2}{n-2}\left(1+u_{n-2}\right) + u_{n-1}$$

$$\Rightarrow u_n = \frac{2}{n-1}\left(1+u_{n-2}\right) + \frac{n-2}{n-1}u_{n-1}$$

$$= \frac{2}{n-1} + \frac{2}{n-1}u_{n-2} + \frac{n}{n-1}u_{n-1} - \frac{2}{n-1}u_{n-1} \tag{4.3}$$

Nous allons montrer le résultat par récurrence. L'égalité est vérifiée pour $n = 2$. On suppose l'égalité vraie jusqu'au rang $n-1$ et on vérifie au rang n. On note $b_n = \sum_{k=0}^{n}\frac{(-2)^k}{k!}$ et $a_k = \frac{(-2)^k}{k!}$. En reprenant la formule 4.3 et en utilisant notre hypothèse d'induction sur u_{n-1}, on obtient :

$$u_n = \frac{2}{n-1} + \frac{2(n-2-(n-2)b_{n-3}-2b_{n-4})}{n-1}$$

$$+\frac{n(n-1-(n-1)b_{n-2}-2b_{n-3})}{n-1} - \frac{2(n-1-(n-1)b_{n-2}-2b_{n-3})}{n-1}u_{n-1}$$

$$= n - b_{n-3}\frac{4n-8}{n-1} + b_{n-2}\frac{3n-n^2-2}{n-1} - \frac{4}{n-1}b_{n-4}$$

Pour montrer le résultat, il suffit de montrer que

$$n - b_{n-3}\frac{4n-8}{n-1} + b_{n-2}\frac{3n-n^2-2}{n-1} - \frac{4}{n-1}b_{n-4} - (n - nb_{n-1} - 2b_{n-2}) = 0$$

or, $b_{n-4} = b_{n-2} - a_{n-2} - a_{n-3}$ et $b_{n-3} = b_{n-2} - a_{n-2}$ donc

$$b_{n-2}\left(\frac{-n^2+n}{n-1}\right) + nb_{n-1} + \frac{1}{n-1}(4n-4)a_{n-2} + \frac{4}{n-1}a_{n-3}$$

$$-nb_{n-2} + nb_{n-1} + \frac{1}{n-1}(4n-4)a_{n-2} + \frac{4}{n-1}a_{n-3}$$

$$-nb_{n-2} + nb_{n-2} + na_{n-1} + \frac{1}{n-1}(4n-4)a_{n-2} + \frac{4}{n-1}a_{n-3}$$

$$na_{n-1} + \frac{1}{n-1}(4n-4)a_{n-2} + \frac{4}{n-1}a_{n-3}$$

$$na_{n-1} - 4\frac{n-1}{2}a_{n-1} + \frac{4}{n-1}\frac{(n-1)(n-2)}{4}a_{n-1}$$

$$= na_{n-1} - 2(n-1)a_{n-1} + (n-2)a_{n-1}$$

$$a_{n-1}(n - 2n + 2 + n - 2) = 0, \text{ ce qui prouve le résultat.}$$

■

Lemme 31 *La limite de l'espérance du rapport d'approximation de* EDGE DELETION *sur P_n lorsque n tend vers l'infini vaut $2 - 2e^{-2}$.*

Preuve. On a

$$\mathbb{E}(ED(P_n)) = n - nb_{n-1} - 2b_{n-2}$$

et

$$\frac{n - nb_{n-1} - 2b_{n-2}}{\frac{n}{2}} \leq \frac{n - nb_{n-1} - 2b_{n-2}}{\lfloor \frac{n}{2} \rfloor} \leq \frac{n - nb_{n-1} - 2b_{n-2}}{\frac{n-1}{2}}$$

On calcule la limite de la partie gauche :

$$\lim_{n\to\infty} \frac{n - nb_{n-1} - 2b_{n-2}}{\frac{n}{2}} = \lim_{n\to\infty} \frac{2n - 2nb_{n-1} - 4b_{n-2}}{n} = 2 - \lim_{n\to\infty} 2b_n - \lim_{n\to\infty} \frac{4b_n}{n} = 2 - 2e^{-2}$$

De la même façon, on calcule la limite de la partie droite :

$$\lim_{n\to\infty} \frac{n - nb_{n-1} - 2b_{n-2}}{\frac{n-1}{2}} = \lim_{n\to\infty} \frac{2n - 2nb_{n-1} - 4b_{n-2}}{n-1} = 2 - \lim_{n\to\infty} 2b_n - \lim_{n\to\infty} \frac{4b_n}{n-1} = 2 - 2e^{-2}$$

On en déduit que l'espérance du rapport d'approximation de EDGE DELETION tend vers $2 - 2e^{-2}$ lorsque la taille du chemin tend vers l'infini. ■

4.5 Expression de l'espérance de Maximum Degree Greedy sur les chemins

Nous avons montré que LISTRIGHT donne de bons résultats sur les chemins. Nous allons montrer que MAXIMUM DEGREE GREEDY, bien que légèrement meilleur, retourne des solutions dont la qualité est très proche de celle de LISTRIGHT. Nous avons le théorème suivant :

Théorème 32 *L'espérance du nombre de sommets retournés par MDG sur P_n vaut :*

$$\mathbb{E}(MDG(P_n)) = \frac{n-1}{2} + \frac{n-1}{2}\sum_{p=0}^{n-1}\frac{(-2)^p}{p!} + \sum_{p=0}^{n-2}\frac{(-2)^p}{p!}$$

Lemme 33 *Une expression récursive de l'espérance du nombre de sommets retournés par MDG sur P_n est :*

$$\mathbb{E}(MDG(P_0)) = \mathbb{E}(MDG(P_1)) = 0 \text{ et } \mathbb{E}(MDG(P_2)) = 1$$

$$\mathbb{E}(MDG(P_n)) = 1 + \frac{2}{n-2}\sum_{k=2}^{n-2}\mathbb{E}(MDG(P_k))$$

Preuve. Maximum Degree Greedy ne sélectionne un sommet que s'il existe au moins une arête et donc au moins 2 sommets. Comme pour les algorithmes ListRight et Edge Deletion, on en déduit que $\mathbb{E}(MDG(P_0)) = \mathbb{E}(MDG(P_1)) = 0$.

À chaque étape de l'algorithme, Maximum Degree Greedy va sélectionner un sommet u_k de degré maximum. On distingue deux cas :

- Le chemin est de taille 2. Maximum Degree Greedy vas donc sélectionner un des deux sommets et ne sélectionnera plus aucun sommet par la suite car le graphe ne contiendra plus d'arête. On en déduit que $\mathbb{E}(MDG(P_2)) = 1$.
- Le chemin est de taille supérieure à 2. Maximum Degree Greedy va donc sélectionner un sommet de degré 2 parmi les $n-2$ sommets de degré 2, c'est-à-dire n'importe quel sommet interne du chemin. Après cette sélection, nous obtenons deux chemins résultants de la suppression de u_k : RP_1 et RP_2. RP_1 est composé des sommets u_1 à u_{k-1}, et RP_2 est composé des sommets u_{k+1} à u_n. Comme la taille du chemin RP_1 est $k-1$ et la taille de RP_2 est $n-k$, nous obtenons que $RP_1 = P_{k-1}$ et $RP_2 = P_{n-k}$.

Le sommet sélectionné par Maximum Degree Greedy est choisi de façon équiprobable parmi les sommets de degré maximum soit $\frac{1}{n-2}$ lorsque $n \geq 3$. Nous pouvons en déduire une expression de l'espérance de la taille de la solution retournée par Maximum Degree Greedy sur P_n :

$$\mathbb{E}(MDG(P_0)) = \mathbb{E}(MDG(P_1)) = 0 \text{ et } \mathbb{E}(MDG(P_2)) = 1$$

$$\mathbb{E}(MDG(P_n)) = \sum_{k=2}^{n-1}\frac{1}{n-2}\cdot(1 + \mathbb{E}\left(MDG\left(P_{k-1}\right) + \mathbb{E}\left(MDG\left(P_{n-k}\right)\right)\right))$$

En décomposant la somme, on obtient :

$$\mathbb{E}(MDG(P_n)) = \sum_{k=2}^{n-1}\frac{1}{n-2} + \sum_{k=2}^{n-1}\frac{1}{n-2}\cdot\mathbb{E}(MDG(P_{k-1}) + \sum_{k=2}^{n-1}\frac{1}{n-2}\cdot\mathbb{E}(MDG(P_{n-k})))$$

On s'aperçoit, en réécrivant les termes des deux dernières sommes qu'elles sont égales :

$$\sum_{k=2}^{n-1}\frac{1}{n-2}\cdot\mathbb{E}(MDG(P_{k-1}) = \frac{1}{n-2}\left(\mathbb{E}(MDG(P_1) + \mathbb{E}(MDG(P_2) + \cdots + \mathbb{E}(MDG(P_{n-2})\right)$$

$$\sum_{k=2}^{n-1}\frac{1}{n-2}\cdot\mathbb{E}(MDG(P_{n-k}))) = \frac{1}{n-2}\left(\mathbb{E}(MDG(P_{n-2}) + \mathbb{E}(MDG(P_{n-3}) + \cdots + \mathbb{E}(MDG(P_1)\right)$$

Nous pouvons reformuler notre expression de manière plus simple :

$$\mathbb{E}(MDG(P_n)) = \sum_{k=2}^{n-1} \frac{1}{n-2} + 2 \cdot \sum_{k=2}^{n-2} \frac{1}{n-2} \cdot \mathbb{E}(MDG(P_k)) \text{ car } \mathbb{E}(MDG(P_1) = 0$$

$$\mathbb{E}(MDG(P_n)) = 1 + \frac{2}{n-2} \sum_{k=2}^{n-2} \mathbb{E}(MDG(P_k))$$

■

Lemme 34 *Pour $n \geq 3$, on a :*

$$\mathbb{E}(MDG(P_n)) = 1 + \frac{2}{n-2} \sum_{k=2}^{n-2} \mathbb{E}(MDG(P_k)) = \frac{n-1}{2} + \frac{n-1}{2} \sum_{p=0}^{n-1} \frac{(-2)^p}{p!} + \sum_{p=0}^{n-2} \frac{(-2)^p}{p!}$$

Preuve. On note $u_n = \mathbb{E}(MDG(P_n))$ pour plus de clarté.

$$\text{Soit } u_n = 1 + \frac{2}{n-2} \sum_{k=2}^{n-2} u_k, \text{ avec } u_0 = u_1 = 0, \text{ et } u_2 = 1.$$

Dans un premier temps, nous allons montrer que $u_n = \frac{n-3}{n-2} u_{n-1} + \frac{2}{n-2} u_{n-2} + \frac{1}{n-2}$. Il suffit de remarquer que :

$$\frac{n-2}{n-3} u_n = \frac{n-2}{n-3} + \frac{2}{n-3} \sum_{k=2}^{n-2} u_k$$

$$= \frac{n-2}{n-3} + \frac{2}{n-3} \sum_{k=2}^{n-3} u_k + \frac{2}{n-3} u_{n-2}$$

$$= 1 + \frac{1}{n-3} + \frac{2}{n-3} \sum_{k=2}^{n-3} u_k + \frac{2}{n-3} u_{n-2}$$

$$= u_{n-1} + \frac{1}{n-3} + \frac{2}{n-3} u_{n-2}$$

d'où

$$u_n = \frac{n-3}{n-2} u_{n-1} + \frac{2}{n-2} u_{n-2} + \frac{1}{n-2}$$

Nous allons maintenant montrer le résultat par récurrence. L'égalité est vraie pour $n = 3$. Nous supposons que l'égalité est vraie jusqu'au rang $n-1$ et nous vérifions au rang n. On a :

$$u_n = \frac{n-3}{n-2} u_{n-1} + \frac{2}{n-2} u_{n-2} + \frac{1}{n-2}$$

et par hypothèse de récurrence, on obtient :

$$u_n = \frac{n-3}{n-2} \left(\frac{n}{2} - 1 + \left(\frac{n}{2} - 1 \right) b_{n-2} + b_{n-3} \right) + \frac{2}{n-2} \left(\frac{n}{2} - \frac{3}{2} + \left(\frac{n}{2} - \frac{3}{2} \right) b_{n-3} + b_{n-4} \right) + \frac{1}{n-2}$$

$$u_n = \frac{(n-3)n}{2(n-2)} - \frac{n-3}{n-2} + \left(\frac{(n-3)n}{2(n-2)} - \frac{n-3}{n-2} \right) b_{n-2} + \frac{n-3}{n-2} b_{n-3} + \frac{n-3}{n-2} + \frac{n-3}{n-2} b_{n-3} + \frac{2}{n-2} b_{n-4} + \frac{1}{n-2}$$

$$u_n = \frac{n-3}{2} + \frac{n-3}{2}b_{n-2} + \frac{n-3}{n-2}b_{n-3} + \frac{n-3}{n-2} + \frac{n-3}{n-2}b_{n-3} + \frac{2}{n-2}b_{n-4} + \frac{1}{n-2}$$

$$u_n = \frac{n-1}{2} + \frac{n-3}{2}b_{n-2} + b_{n-3}\frac{n-3}{n-2}2 + \frac{2}{n-2}b_{n-4}$$

Pour prouver le résultat, il suffit de montrer que

$$Z = \frac{n-1}{2} + \frac{n-3}{2}b_{n-2} + b_{n-3}\frac{n-3}{n-2}2 + \frac{2}{n-2}b_{n-4} - \left(\frac{n-1}{2} + \frac{n-1}{2}b_{n-1} + b_{n-2}\right) = 0$$

En développant, on obtient :

$$Z = \frac{n-5}{2}b_{n-2} + b_{n-3}\frac{n-3}{n-2}2 + \frac{2}{n-2}b_{n-4} - \frac{n-1}{2}b_{n-1}$$

En remarquant que $b_{n-k} = b_n - \sum_{j=n-k+1}^{n} a_j$, on trouve :

$$Z = \frac{n-5}{2}\left(b_{n-1} - a_{n-1}\right) + \left(b_{n-1} - a_{n-1} - a_{n-2}\right)\frac{n-3}{n-2}2 + \frac{2}{n-2}\left(b_{n-1} - a_{n-1} - a_{n-2} - a_{n-3}\right) - \frac{n-1}{2}b_{n-1}$$

$$Z = b_{n-1}\left(\frac{n-5}{2} + \frac{n-3}{n-2}2 + \frac{2}{n-2} - \frac{n-1}{2}\right) - a_{n-1}\left(\frac{n-5}{2} + \frac{n-3}{n-2}2 + \frac{2}{n-2}\right)$$
$$-a_{n-2}\left(\frac{n-3}{n-2}2 + \frac{2}{n-2}\right) - \frac{2}{n-2}a_{n-3}$$
$$Z = \frac{n-1}{2}a_{n-1} - 2a_{n-2} - \frac{2}{n-2}a_{n-3}$$

En remarquant que $a_{n-1} = -\frac{n}{2}a_n$, on obtient :

$$Z = \frac{n-1}{2}a_{n-1} - 2\left(-\frac{n-1}{2}a_{n-1}\right) - \frac{2}{n-2}\left(\frac{(n-1)(n-2)}{4}a_{n-1}\right)$$
$$Z = a_{n-1}\left(-\frac{n-1}{2} + n - 1 - \frac{n-1}{2}\right) = 0$$

ce qui montre le résultat. ■

Lemme 35 *La limite de l'espérance du rapport d'approximation de* Maximum Degree Greedy *sur P_n lorsque n tend vers l'infini vaut $1 + e^{-2}$.*

Preuve.

$$\text{On a } \mathbb{E}(MDG(P_n)) = \frac{n-1}{2} + \frac{n-1}{2}b_{n-1} + b_{n-2}$$
$$\text{et } \mathbb{E}(LR(P_n)) = \frac{n-1}{2} + \frac{n+1}{2}b_n + b_{n-1}$$

Nous allons montrer que la limite, lorsque n tend vers l'infini, de la différence entre $\mathbb{E}(MDG(P_n))$ et $\mathbb{E}(LR(P_n))$ tend vers 0 et donc que le rapport d'approximation de Maximum Degree Greedy sur les chemins tend vers $1 + e^{-2}$ (par le lemme 27).

$$\mathbb{E}(LR(P_n)) - \mathbb{E}(MDG(P_n)) = \frac{n-1}{2} + \frac{n+1}{2}b_n + b_{n-1} - \frac{n-1}{2} - \frac{n-1}{2}b_{n-1} - b_{n-2}$$

$$= \frac{n+1}{2}b_n + b_{n-1} - \frac{n-1}{2}b_{n-1} - b_{n-2}$$
$$= \frac{n+1}{2}b_n - \frac{n-1}{2}b_{n-1} + b_{n-1} - b_{n-2}$$
$$= \frac{n-1}{2}a_n + a_{n-1}$$

Comme $a_n = \frac{-2}{n}a_{n-1}$ on obtient

$$\mathbb{E}(LR(P_n)) - \mathbb{E}(MDG(P_n)) = -\frac{n-1}{n}a_{n-1} + a_{n-1}$$

et comme a_{n-1} tend vers 0 lorsque n tend vers l'infini, on obtient le résultat. ■

4.6 Répartition expérimentale des solutions retournées par les algorithmes étudiés

Dans cette partie, nous allons plus loin dans l'étude du comportement de nos algorithmes sur les chemins en observant expérimentalement la manière dont sont réparties les tailles des solutions retournées. Comme nous avons vu que GIC retourne toujours une solution optimale et que l'algorithme DFS retourne pratiquement tous les sommets, nous nous focalisons uniquement sur les algorithmes ListLeft, ListRight, Edge Deletion et Maximum Degree Greedy.

Les figures 4.1, 4.2, 4.3 et 4.4 sont les résultats de 100000 exécutions de chacun de ces 4 algorithmes sur un chemin de taille 100. Les méthodes expérimentales utilisées sont décrites dans le chapitre 5, qui est consacré à une étude expérimentale (ne se limitant pas aux chemins) des différents algorithmes. Pour chaque figure, nous avons affiché sous forme d'histogramme la répartition des tailles des solutions, et nous avons tracé deux lignes verticales correspondant respectivement à la taille de la solution optimale et à l'espérance analytique obtenue précédemment.

On peut tout d'abord constater que les résultats précédents sont parfaitement cohérents avec les données expérimentales. En effet, si on note $\mathbb{E}_{exp}(A(G))$ l'espérance expérimentale de la taille des solutions retournées par un algorithme A sur un graphe G, on a :

- $\mathbb{E}(LL(P_{100})) \approx 66.3$ et $\mathbb{E}_{exp}(LL(P_{100})) \approx 66.05$
- $\mathbb{E}(LR(P_{100})) \approx 56.46$ et $\mathbb{E}_{exp}(LR(P_{100})) \approx 56.45$
- $\mathbb{E}(ED(P_{100})) \approx 86.19$ et $\mathbb{E}_{exp}(ED(P_{100})) \approx 86.29$
- $\mathbb{E}(MDG(P_{100})) \approx 56.33$ et $\mathbb{E}_{exp}(MDG(P_{100})) \approx 56.33$

Sur ces figures, on remarque aussi que les répartitions suivent une loi en forme de cloche centrée sur l'espérance, ce qui suggère que nous pouvons approximer cette loi par une loi normale dont la fonction de densité est

$$\varphi(x) = \frac{1}{\sqrt{\sigma 2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \text{ avec } \sigma \text{ la variance et } \mu \text{ l'espérance}$$

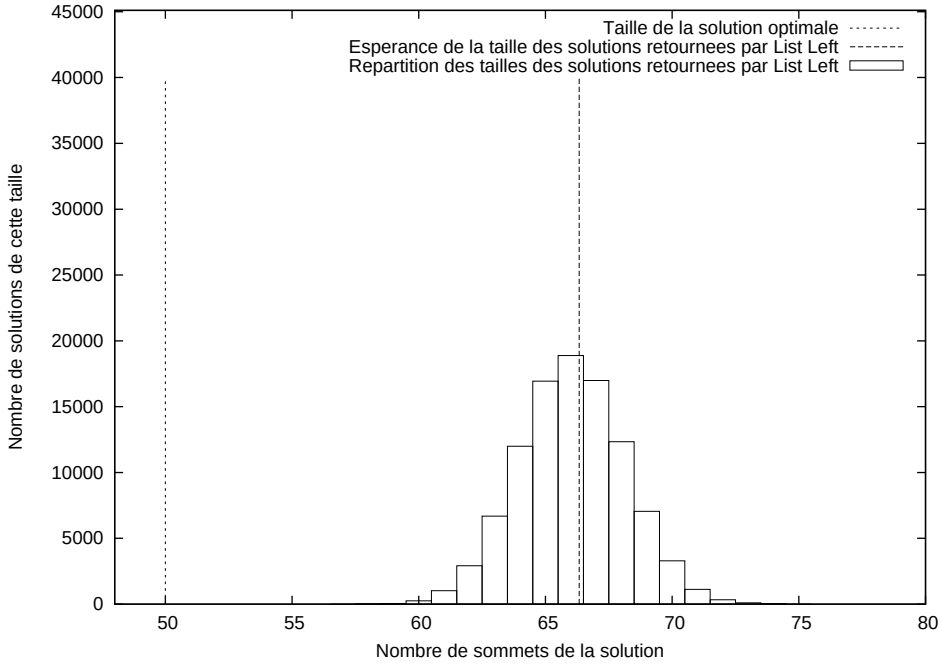


FIG. 4.1 – Répartition expérimentale des tailles des solutions retournées par LISTLEFT sur un chemin de taille 100 après 100000 itérations.

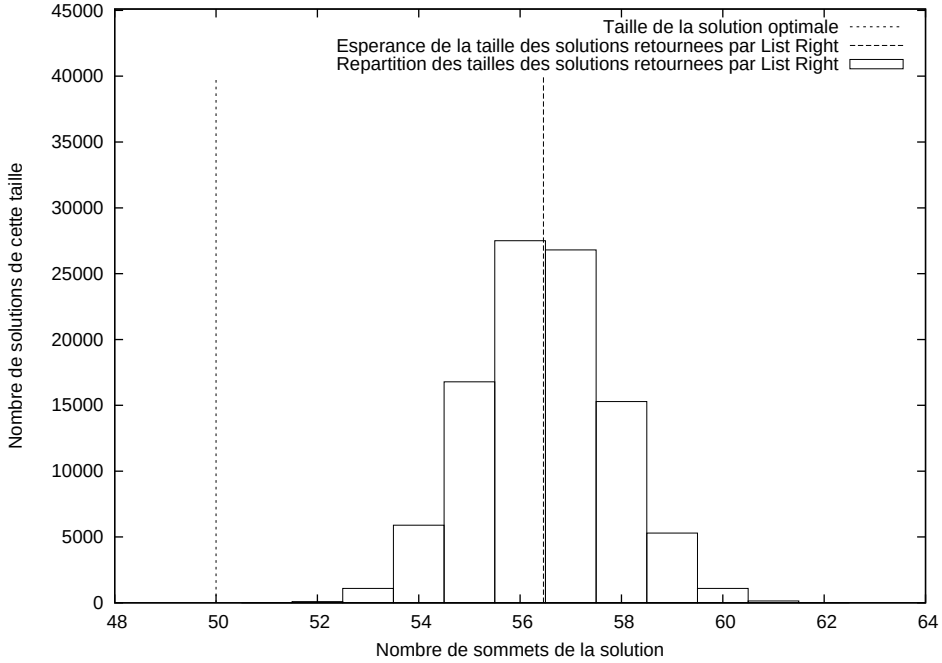


FIG. 4.2 – Répartition expérimentale des tailles des solutions retournées par LISTRIGHT sur un chemin de taille 100 après 100000 itérations

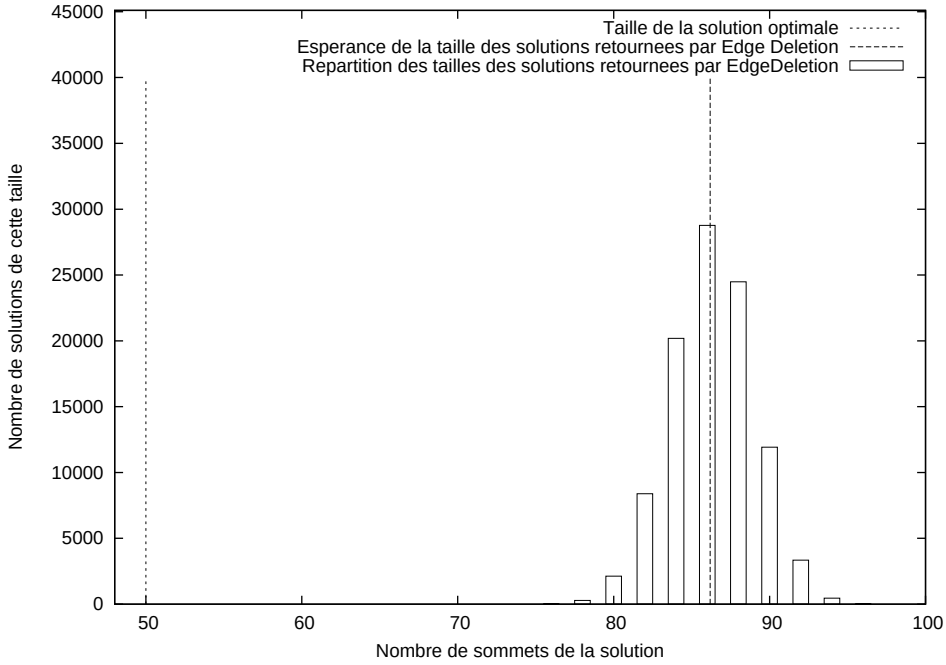


FIG. 4.3 – Répartition expérimentale des tailles des solutions retournées par EDGE DELETION sur un chemin de taille 100 après 100000 itérations.

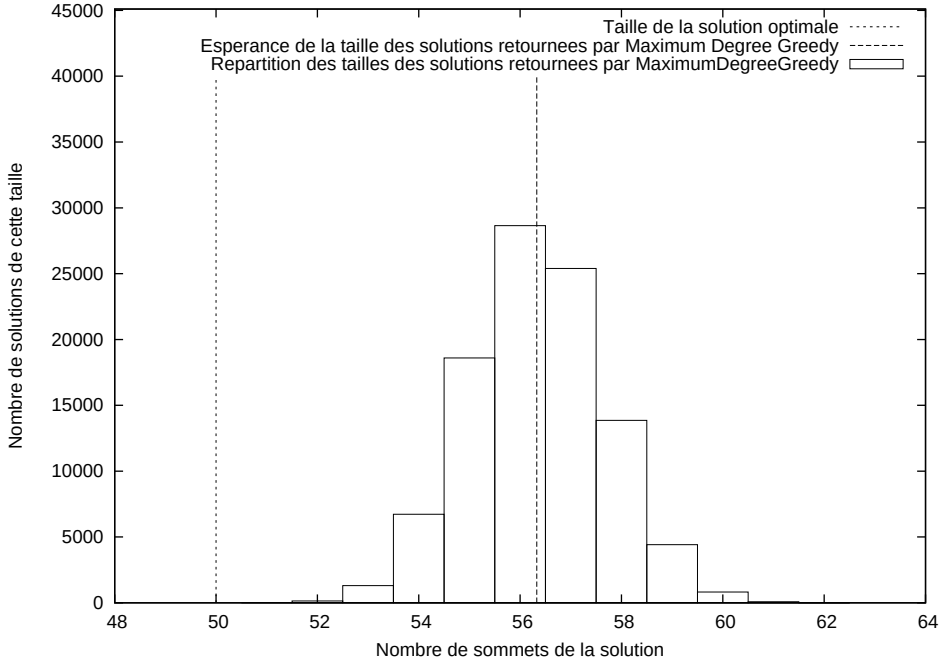


FIG. 4.4 – Répartition expérimentale des tailles des solutions retournées par MAXIMUM DEGREE GREEDY sur un chemin de taille 100 après 100000 itérations.

Pour déterminer la loi de répartition pour chacun des 4 algorithmes, nous avons besoin de l'espérance (que nous connaissons de façon théorique) et de la variance, que nous allons déterminer expérimentalement. La figure 4.5 représente l'évolution de la variance expérimentale des 4 algorithmes en fonction de la taille du chemin. Très rapidement la variance devient (pour chaque algorithme) linéaire par rapport au nombre de sommets du chemin. On peut donc facilement déterminer une équation de droite qui permettra de prédire la variance.

Remarque. Les algorithmes ListRight et Maximum Degree Greedy possèdent non seulement une espérance très proche, mais leur variance expérimentale est quasiment identique ce qui montre, encore une fois, que ces deux algorithmes ont un comportement très similaire sur les chemins.

La figure 4.6 représente, comme la figure 4.5, l'évolution de la variance pour chacun des 4 algorithmes, mais nous avons rajouté les droites obtenues afin de montrer leur cohérence avec nos résultats expérimentaux. Ainsi, pour un chemin de taille n, la variance (prédite) sera de :

- $0.0183n + 0.03$ pour ListRight et Maximum Degree Greedy
- $0.0445n - 0.05$ pour ListLeft
- $0.0733n + 0.15$ pour Edge Deletion

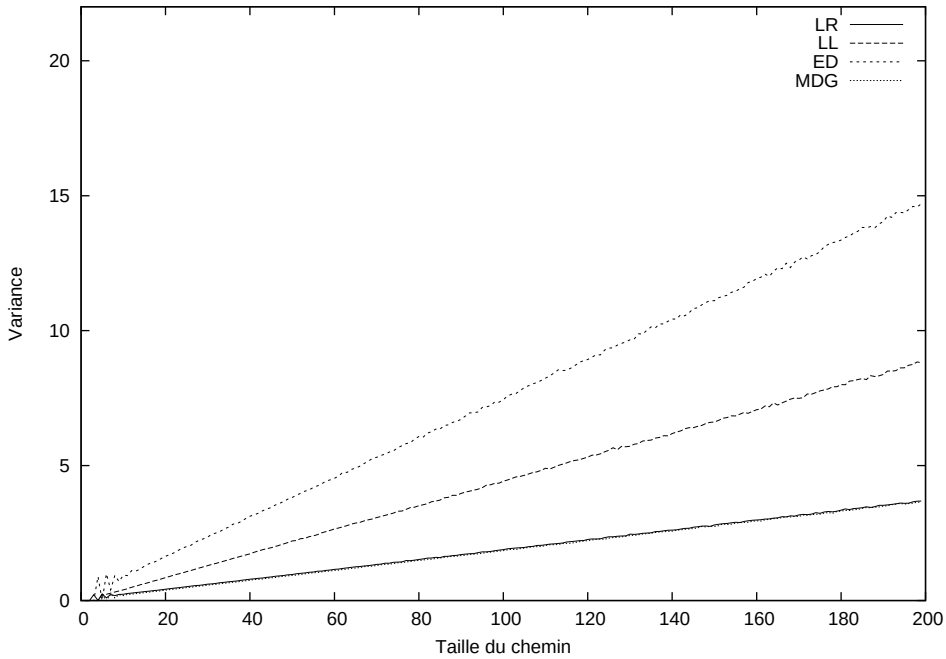


Fig. 4.5 – Évolution de la variance en fonction de la taille du chemin pour les algorithmes Edge Deletion, Maximum Degree Greedy, ListRight et ListLeft

Nous pouvons donc déterminer les fonctions de densités des différents algorithmes en remplaçant σ et μ par leur valeur correspondante. Les figures 4.7, 4.8, 4.9 et 4.10 montrent pour chaque algorithme la prévision de répartition des tailles de solutions ainsi que la répartition obtenue expérimentalement sur un chemin de taille 1000 après 10000 itérations.

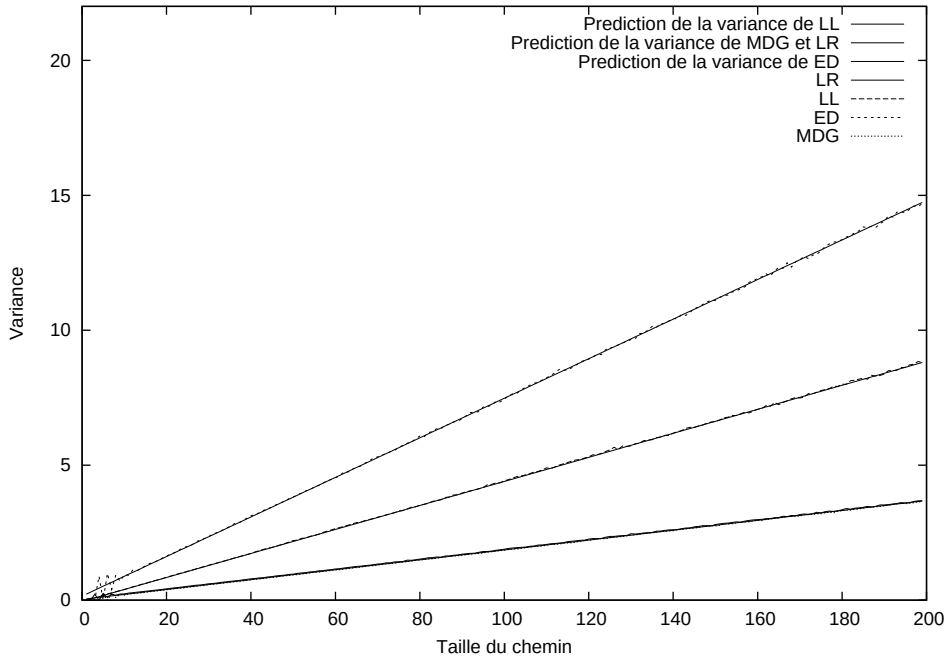


FIG. 4.6 – Prédiction de l'évolution de la variance en fonction de la taille du chemin pour les algorithmes EDGE DELETION, MAXIMUM DEGREE GREEDY, LISTRIGHT et LISTLEFT

Remarque. Pour obtenir la fonction de densité de l'algorithme EDGE DELETION, nous avons dû procéder un peu différemment. En effet, cet algorithme ne retourne que des solutions de taille paire, et l'approximation de sa fonction de densité, qui est continue, ne prend pas en compte le fait qu'on ne peut pas obtenir de solution de taille impaire. Nous avons donc divisé par 2 toutes les solutions retournées par EDGE DELETION, ce qui nous permet d'obtenir une distribution du même type que pour les algorithmes précédents (notons qu'en effectuant cette manipulation, nous divisons l'espérance par 2 et la variance par 4). Nous avons ensuite calculé sa fonction de densité pour approximer la répartition des données et nous avons tracé la courbe en faisant correspondre une abscisse de k de la fonction de densité ainsi obtenue à une abscisse de $2k$ pour la fonction de densité qui doit approcher dans notre répartition initiale.

4.7 Bilan de l'évaluation du comportement des algorithmes sur les chemins

Nous avons évalué analytiquement l'espérance de 6 algorithmes sur les chemins : 2 algorithmes 2−approché (EDGE DELETION et DEPTH FIRST SEARCH), 2 algorithmes de liste (LISTLEFT et LISTRIGHT) et 2 algorithmes gloutons possédant un mauvais rapport d'approximation en pire cas (MAXIMUM DEGREE GREEDY et GREEDY INDEPENDENT COVER). Nous avons mis en évidence que les algorithmes avec un rapport d'approximation en pire cas de 2 sont ceux qui obtiennent les plus mauvais résultats. Les chemins étant des graphes dont toute solution (à un sommet près) est au plus 2−approchée, ces algorithmes se retrouvent sur un pied d'égalité avec les autres algorithmes (en terme de rapport d'approximation en pire cas) et sont pénalisés par la structure des solutions

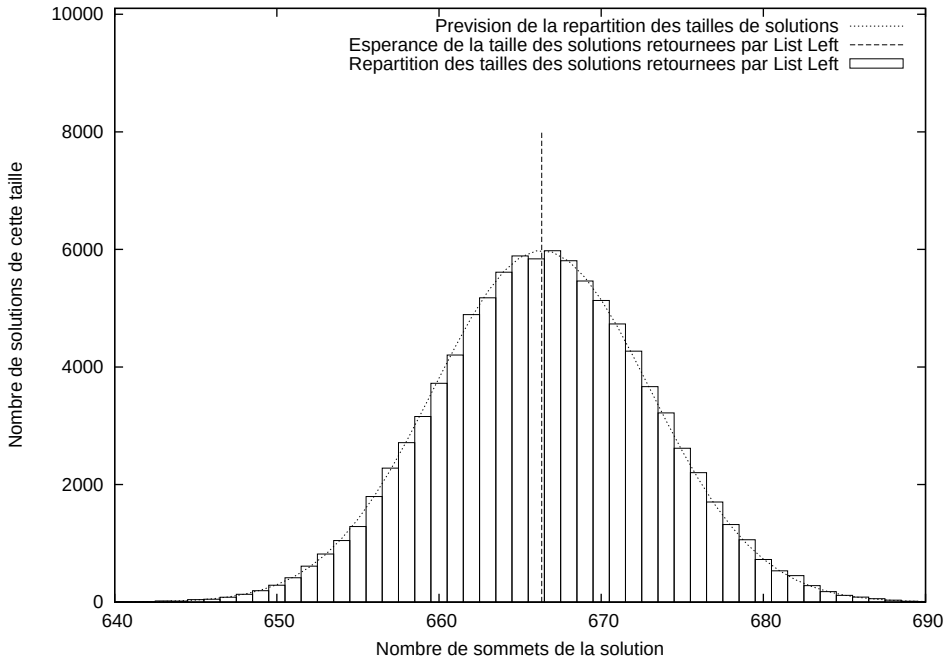


FIG. 4.7 – Prédiction de la répartition des tailles des solutions retournées par LISTLEFT sur un chemin de taille 100.

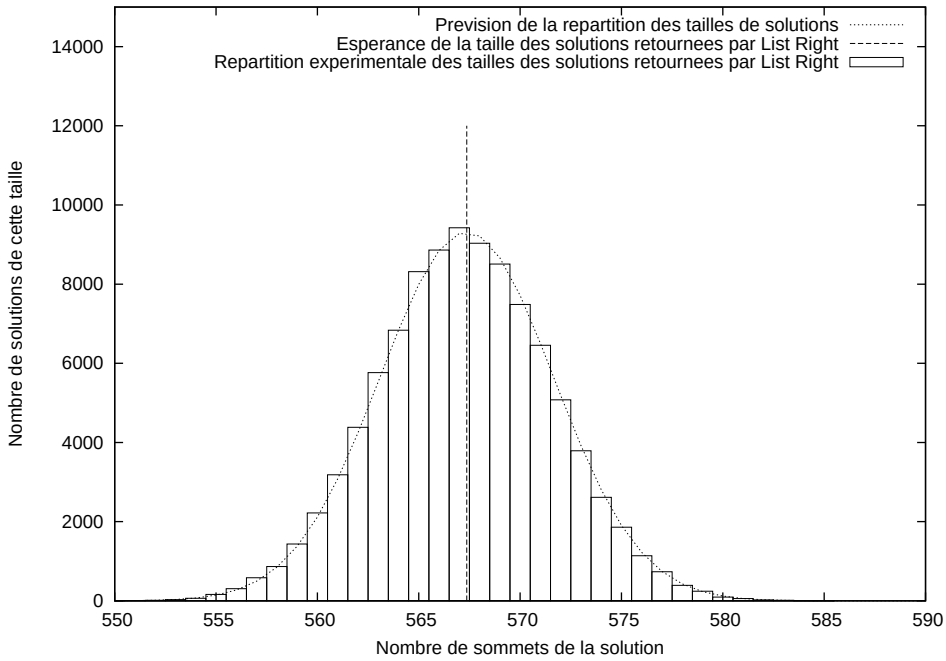


FIG. 4.8 – Prédiction de la répartition des tailles des solutions retournées par LISTRIGHT sur un chemin de taille 100.

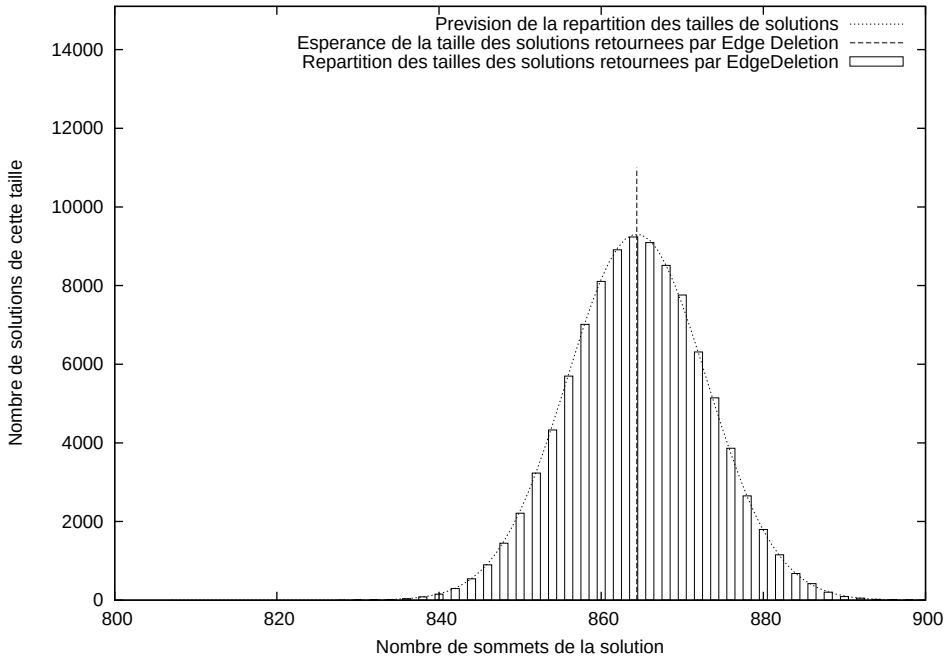


FIG. 4.9 – Prédiction de la répartition des tailles des solutions retournées par EDGE DELETION sur un chemin de taille 100.

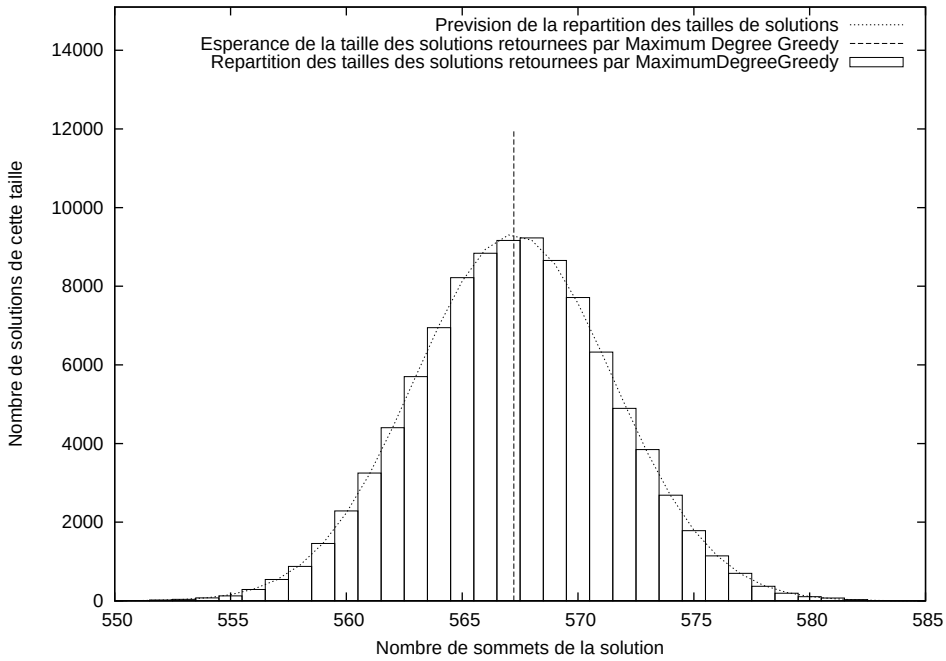


FIG. 4.10 – Prédiction de la répartition des tailles des solutions retournées par MAXIMUM DEGREE GREEDY sur un chemin de taille 100.

qu'ils retournent. En effet, les sommets d'un couplage ou d'un arbre de parcours en profondeur privé de ses feuilles sont rarement des solutions minimales pour l'inclusion. L'algorithme Greedy Independent Cover profite au contraire de son heuristique qui est optimale sur les arbres, et donc sur les chemins. On obtient, à partir de nos résultats analytiques le classement suivant en fonction de l'espérance de la taille des solutions retournées :

1. Greedy Independent Cover
2. Maximum Degree Greedy
3. ListRight
4. ListLeft
5. Edge Deletion
6. Depth First Search

Le tableau 4.1 propose un bilan des résultats théoriques obtenus.

De plus, nous avons obtenu de manière expérimentale la variance des tailles des solutions retournées. Une variance faible indique que les tailles des solutions seront plutôt concentrées autour de l'espérance, ce qui permet d'apporter une garantie plus ou moins forte sur la qualité de la solution. Les algorithmes Greedy Independent Cover et Depth First Search retournent toujours la même taille de solution. Les 4 autres algorithmes peuvent être classés de la façon suivante (de la variance la plus faible à la plus forte) :

1. Maximum Degree Greedy et ListRight
2. ListLeft
3. Edge Deletion

Bien que Edge Deletion possède la variance la plus élevée des 6 algorithmes étudiés, il n'est pas capable de retourner de bonnes solutions. En effet, dans [Del06], il a été montré le résultat suivant :

$$Worst(MDG(P_n)) \leq Best(ED(P_n))$$

avec $Worst(A(G))$ (Resp. $Best(A(G))$) la taille de la plus mauvaise (Resp. meilleure) solution qu'un algorithme A peut retourner lorsqu'il est appliqué sur le graphe G. Ce résultat, associé à celui de l'espérance de Depth First Search, montre que Maximum Degree Greedy domine les deux algorithmes. Comme l'algorithme ListRight possède des performances très similaires à celles de Maximum Degree Greedy, nous en concluons que ListRight domine (du moins statistiquement) les deux algorithmes 2−approchés.

Pour les 4 algorithmes ayant une variance non nulle, nous avons aussi fourni la loi de répartition des tailles des solutions, nous permettant ainsi de calculer la probabilité qu'un algorithme retourne une solution de taille x.

Algorithme	Espérance	Limite de l'espérance du rapport d'approximation
Greedy Independent Cover	$\lfloor \frac{n}{2} \rfloor = \lvert OPT \rvert$	1
Maximum Degree Greedy	$\frac{n-1}{2} + \frac{n-1}{2} \sum_{p=0}^{n-1} \frac{(-2)^p}{p!} + \sum_{p=0}^{n-2} \frac{(-2)^p}{p!}$	$1 + e^{-2} \approx 1.13$
ListRight	$\frac{n-1}{2} + \frac{n+1}{2} \cdot \sum_{k=0}^{n} \frac{(-2)^k}{k!} + \sum_{k=0}^{n-1} \frac{(-2)^k}{k!}$	$1 + e^{-2} \approx 1.13$
ListLeft	$\frac{2n-1}{3}$	$\frac{4}{3} \approx 1.33$
Edge Deletion	$n - n \sum_{p=0}^{n-1} \frac{(-2)^p}{p!} - 2 \sum_{p=0}^{n-2} \frac{(-2)^p}{p!}$	$2 - 2e^{-2} \approx 1.73$
Depth First Search	$n - 2 + \frac{2}{n}$	2

Tab. 4.1 – Tableau récapitulatif des résultats analytiques obtenus sur les chemins.

Annexe du chapitre

Dans la section 4.3 et dans un soucis de lisibilité, nous avons donné une preuve permettant de vérifier la validité du théorème 24 sans pour autant expliquer comment nous avons trouvé le résultat. Cette annexe contient la preuve constructive de notre théorème. La méthode utilisée est la suivante :

Méthode de résolution. Soit u_n une suite définie récursivement, et $F(X) = \sum_{n\geq 0} u_n X^n$ sa fonction génératrice. Nous allons exprimer $F(X)$ en fonction de $F'(X)$ de manière à obtenir une équation différentielle. En résolvant cette équation différentielle, nous obtenons une formulation alternative de $F(X)$ et par un jeu de réécriture impliquant certaines fonctions génératrices ordinaires usuelles, on obtient $F(X) = \sum_{n\geq 0} v_n X^n$, avec v_n une suite définie de manière non récursive. Par définition, on a $u_n = v_n$, obtenant ainsi une expression non récursive de u_n.

Cette méthode permet à la fois de trouver et de prouver la formulation close de l'espérance. Le déroulement de cette méthode est donné pour l'algorithme ListRight mais elle est identique pour les algorithmes Edge Deletion et Maximum Degree Greedy.

Rappel du théorème 24 : l'espérance du nombre de sommets retournés par ListRight sur P_n vaut :

$$\mathbb{E}(LR(P_n)) = \frac{n-1}{2} + \frac{n+1}{2} \cdot \sum_{k=0}^{n} \frac{(-2)^k}{k!} + \sum_{k=0}^{n-1} \frac{(-2)^k}{k!}$$

Preuve. Dans un soucis de lisibilité, on notera $\mathbb{E}(LR(P_n)) = u_n$. On a :

$$u_0 = 0, u_1 = 0, u_n = \frac{2}{n} \cdot \left(n - 1 + \sum_{i=0}^{n-2} u_i \right)$$

Soit $F(X)$ la fonction génératrice de u_n :

$$F(X) = \sum_{n\geq 0} u_n X^n$$

Nous allons exprimer $F(X)$ en fonction de $F'(X)$. Par définition, nous avons

$$F'(X) = \sum_{n\geq 2} n u_n X^{n-1} \quad \text{car } u_0 = u_1 = 0$$

En remplaçant u_n par son expression, nous obtenons

$$F'(X) = 2 \sum_{n\geq 2} (n-1) X^{n-1} + 2 \sum_{n\geq 2} \left(\sum_{i=0}^{n-2} u_i \right) X^{n-1} \tag{4.4}$$

Comme $\frac{1}{1-X} = \sum_{n\geq 0} X^n$ (voir [Wil06] pour plus de détails sur les séries usuelles), nous avons

$$\frac{1}{(1-X)^2} = \sum_{n\geq 1} n X^{n-1} \text{ et donc } \sum_{n\geq 1} n X^n = \frac{X}{(1-X)^2} \tag{4.5}$$

On rappelle l'opération de convolution sur les fonctions (séries) génératrices ordinaires :

$$\sum_{n\geq 0}(a_nX^n)\cdot\sum_{n\geq 0}(b_nX^n)=\sum_{n\geq 0}\left(\sum_{0\leq k\leq n0}(a_nb_{n-k})\right)X^n$$

De plus, comme $\frac{1}{1-X}=\sum_{n\geq 0}X^n$ et $F(X)=\sum_{k\geq 0}u_kX^k$, on a

$$\begin{aligned}\frac{X}{1-X}\cdot F(X) &= X\cdot\sum_{n\geq 0}\left(\sum_{k=0}^{n}u_k\right)X^n &(4.6)\\ &= X\cdot\sum_{n\geq 2}\left(\sum_{k=0}^{n-2}u_k\right)X^{n-2} &(4.7)\\ &= \sum_{n\geq 2}\left(\sum_{k=0}^{n-2}u_k\right)X^{n-1} &(4.8)\end{aligned}$$

En reportant 4.5 et 4.8 dans 4.4 , on obtient :

$$F'(X)=\frac{2X}{(1-X)^2}+\frac{2X}{1-X}F(X) \tag{4.9}$$

Nous allons maintenant résoudre cette équation différentielle en utilisant la méthode de variation des constantes. Cette méthode permet de déterminer les solutions d'une équation différentielle avec second membre, connaissant les solutions de l'équation homogène (c'est-à-dire sans second membre).

Nous allons donc résoudre l'équation homogène :

$$F'(X)=\frac{2X}{1-X}F(X)$$
$$\Leftrightarrow\frac{F'(X)}{F(X)}=-2+\frac{2}{1-X}$$

Comme $(\ln(F(X)))'=\dfrac{F'(X)}{F(X)}$ on obtient en intégrant que $\ln(F(X))=-2X-2\ln(1-X)+k$

$$\text{Nous en déduisons que } F(X)=\frac{K}{(1-X)^2}\cdot e^{-2X}$$

On cherche des solutions pour 4.9 qui sont de la forme de $F(X)=\frac{K(X)}{(1-X)^2}\cdot e^{-2X}$. Ainsi,

$$F'(X)=\frac{K'(X)}{(1-X)^2}e^{-2X}+\frac{2K(X)}{(1-X)^3}e^{-2X}-2\frac{K(X)}{(1-X)^2}e^{-2X}$$

$$=\frac{K'(X)}{(1-X)^2}e^{-2X}+\frac{2K(X)}{(1-X)^2}e^{-2X}\left(\frac{1}{1-X}-1\right)$$

$$=\frac{K'(X)}{(1-X)^2}e^{-2X}+\frac{2XK(X)}{(1-X)^3}e^{-2X}$$

$$= \frac{K'(X)}{(1-X)^2}e^{-2X} + 2\frac{X}{(1-X)}F(X)$$

Par conséquent,

$$\frac{K'(X)}{(1-X)^2}e^{-2X} = \frac{2X}{(1-X)^2}$$

$$K'(x) = 2Xe^{2X}$$

$$K(X) = \frac{1}{2}(2X-1)e^{2X} + k = (X-1)e^{2X} + \frac{1}{2}e^{2X} + k$$

Finalement,

$$F(X) = -\frac{1}{1-X} + \frac{1}{2}\frac{1}{(1-X)^2} + \frac{k}{(1-X)^2}e^{-2X} \tag{4.10}$$

Nous pouvons déterminer la valeur de k : $F(0) = u_0 = 0 \Rightarrow -1 + \frac{1}{2} + k = 0 \Rightarrow k = \frac{1}{2}$

$$F(X) = \frac{1}{2}\frac{1}{(1-X)^2} - \frac{1}{1-X} + \frac{1}{2}\frac{1}{(1-X)^2}e^{-2X}$$

En utilisant les séries génératrices ordinaires usuelles suivantes, nous allons déterminer une nouvelle expression de notre série génératrice de départ :

- $\frac{1}{(1-X)^2} = \sum_{n\geq 1} nX^{n-1} = \sum_{n\geq 0}(n+1)X^n$
- $\frac{1}{1-X} = \sum_{n\geq 0} X^n$
- $\frac{1}{(1-X)^2}e^{-2X} = \left(\sum_{n\geq 0}(n+1)X^n\right)\left(\sum_{p\geq 0}\frac{(-2)^p}{p!}X^p\right)$
- $\frac{1}{(1-X)^2}e^{-2X} = \sum_{n\geq 0}\left(\sum_{p=0}^{n}\frac{(-2)^p}{p!}(n-p+1)\right)X^n$

Soit $b_n = \sum_{p=0}^{n}\frac{(-2)^p}{p!}$, une somme partielle de $\sum_0^\infty \frac{(-2)^p}{p!} = e^{-2}$. Ainsi,

$$\begin{aligned}
\sum_{p=0}^{n}\frac{(-2)^p}{p!}(n-p+1) &= (n+1)b_n - \sum_{p=0}^{n} p\frac{(-2)^p}{p!} && (4.11)\\
&= (n+1)b_n - \sum_{p=1}^{n}\frac{(-2)^p}{(p-1)!} && (4.12)\\
&= (n+1)b_n - \sum_{k=0}^{n-1}\frac{(-2)^{k+1}}{k!} && (4.13)\\
&= (n+1)b_n + 2b_{n-1} && (4.14)
\end{aligned}$$

ce qui nous conduit à

$$F(X) = \sum_{n\geq 0}\left(\frac{n-1}{2} + \frac{n+1}{2}b_n + b_{n-1}\right)X^n$$

et comme $F(X) = \sum_{n\geq 0} u_n X^n$, nous obtenons finalement une formule close de u_n :

$$u_n = \frac{n-1}{2} + \frac{n+1}{2}b_n + b_{n-1} \tag{4.15}$$

■

Chapitre 5

Comparaison expérimentale

Dans les chapitres précédents, nous avons mené une étude comparative théorique du comportement de différents algorithmes pour le problème du vertex cover. Les résultats théoriques peuvent être difficiles à obtenir ou, tout du moins, prennent du temps à être découverts. Pour aller plus loin, nous avons décidé de mener une comparaison expérimentale de la qualité moyenne des algorithmes précédemment utilisés. Dans un premier temps, nous allons déterminer la mesure qui nous semble la plus adéquate pour capturer la qualité moyenne des algorithmes. Dans un deuxième temps, nous allons définir la méthode que nous avons suivie (par exemple la façon que nous avons eue de gérer l'aléatoire) et enfin, nous présenterons les résultats obtenus sur différents échantillons de graphes.

5.1 Que faut il mesurer ?

Lorsque l'on effectue une batterie de tests expérimentaux, il convient de déterminer au préalable le phénomène que l'on souhaite capturer. Dans cette thèse, notre objectif est de comparer au mieux (et donc qualitativement) les performances de différents algorithmes d'approximation qui offrent une garantie sur leurs performances en pire cas.

Pour un problème de minimisation tel que le vertex cover et pour un ensemble d'instances donné, la manière la plus évidente pour comparer deux algorithmes est de comparer les tailles des solutions qu'ils retournent. Ces performances brutes vont nous permettre, par exemple, d'établir un classement des différents algorithmes sur cet ensemble d'instances.

Cependant, cette mesure brute n'est pas totalement satisfaisante car elle ne nous renseigne pas, *a priori*, sur la qualité des solutions retournées. Supposons par exemple qu'un algorithme retourne une solution de taille 20. Suivant que la taille de la solution optimale est de 20 ou de 1, notre regard sur la performance qualitative de cet algorithme va changer du tout au tout, indépendamment d'une éventuelle comparaison avec les performances d'un autre algorithme.

Afin d'évaluer la qualité des solutions retournées par un algorithme, on utilise généralement le rapport d'approximation « classique » qui compare la taille de la solution considérée et la taille de la solution optimale. En accord avec notre besoin de garantie sur la qualité des solutions, l'algorithme devrait être jugé sur ses performances en pire cas. Comme nous l'avons vu dans les chapitres précédents, l'évaluation en pire cas induit de nombreux biais car elle ne reflète généralement pas les performances globales de l'algorithme (par exemple l'algorithme Maximum Degree Greedy

est presque optimal en moyenne sur les graphes Anti-MDG alors qu'en pire cas ses performances sont désastreuses).

Dans notre étude, nous utiliserons donc comme point de repère la valeur de la solution optimale pour comparer qualitativement les solutions retournées par les différents algorithmes, mais nous n'utiliserons pas l'évaluation en pire cas, car elle est trop peu représentative des performances globales d'un algorithme.

Toutefois, comparer la taille d'une solution à la taille de la solution optimale n'est pas totalement satisfaisant. En utilisant cette mesure, deux solutions ayant un rapport d'approximation très proches seront considérées de même qualité. Dans le cas d'un graphe complet à n sommets, il n'existe que deux tailles de solutions : n et $n-1$. Cela signifie que toutes les solutions auront un comportement équivalent, que ce soit du point de vue des performances brutes que du rapport d'approximation. Cependant, on peut voir les choses sous un autre angle : étant donné que la très grande majorité des algorithmes retournent une solution de taille $n-1$ (et donc optimale), comme par exemple Greedy Independent Cover, Maximum Degree Greedy, ListRight, ListLeft et Depth First Search, et que seul l'algorithme Edge Deletion (parmi ceux que nous étudions dans cette thèse) va retourner n sommets (et uniquement lorsque n est pair), on peut considérer que Edge Deletion possède un comportement anormal, même s'il ne s'agit que d'un sommet supplémentaire, car il retourne la totalité des sommets, et donc la pire solution possible. Il convient donc de prendre comme référence supplémentaire la valeur de la pire solution, ce qui va nous permettre de situer une solution sur l'intervalle de toutes les solutions possibles. Le rapport d'approximation différentiel (voir [MPT03] pour une présentation détaillée) a justement pour objectif de prendre en compte la valeur de la solution retournée, la taille de la solution optimale et la taille de la plus mauvaise solution pouvant être retournée. Plus précisément, l'approximation différentielle mesure le chemin parcouru depuis la pire solution pour une instance I sur l'ensemble des valeurs possibles (c'est-à-dire l'intervalle défini par la taille de la pire solution (notée $\omega(I)$) moins la taille de la solution optimale (notée $\beta(I)$)). La qualité d'une solution S en approximation différentielle est donnée par le rapport $\frac{\omega(I)-|S|}{\omega(I)-\beta(I)}$. Plus ce rapport est proche de 1, plus la solution est éloignée de la pire solution et donc proche de la solution optimale. Au contraire, un rapport d'approximation différentiel proche de 0 dénote une solution de mauvaise qualité. En reprenant l'exemple d'un graphe complet à n sommets, une solution de taille $n-1$ obtiendra un rapport de 1 tandis qu'une solution de taille n obtiendra un rapport de 0, capturant ainsi le comportement anormal de Edge Deletion.

Prendre en compte à la fois la valeur de la solution optimale et la valeur de la pire solution pour déterminer la qualité d'une solution est donc, *a priori*, une bonne façon de procéder.

L'erreur est une mesure équivalente à l'approximation différentielle. Cette mesure n'indique pas la distance de la solution par rapport à la pire solution possible, mais par rapport à la solution optimale. On peut ainsi déterminer le rapport à l'erreur : $\frac{|S|-\beta(I)}{\omega(I)-\beta(I)}$. Le rapport à l'erreur et le rapport d'approximation différentiel capturent les mêmes informations dans le sens où le rapport à l'erreur est égal à 1 moins le rapport d'approximation différentiel. C'est ce rapport à l'erreur que nous allons utiliser pour comparer les performances des différents algorithmes. Nous allons toutefois modifier cette mesure pour prendre en compte la diversité des solutions pouvant être retournées par

un même algorithme. Ainsi, nous allons calculer la distance de la moyenne des solutions par rapport à la taille de la solution optimale, ce qui nous donne le rapport suivant : $\frac{\mathbb{E}(A(I))-\beta(I)}{\omega(I)-\beta(I)}$. Nous allons encore apporter une modification à cette mesure pour prendre en compte le fait que notre démarche est expérimentale. En effet, suivant le nombre d'itérations d'un même algorithme sur une instance I, la moyenne expérimentale des tailles des solutions sera plus ou moins proche de l'espérance théorique de la taille des solutions retournées par cet algorithme. Cette marge d'incertitude ne nous permettra pas de déterminer de façon exacte si un algorithme est réellement meilleur qu'un autre, mais si deux algorithmes ont des performances expérimentales très proches, nous pouvons être certains que ces deux algorithmes vont retourner des solutions qui sont d'une qualité comparable. Ainsi, nous allons discrétiser notre rapport à l'erreur, ce qui nous permettra d'une part, d'absorber ce bruit inhérent à toute expérimentation et, d'autre part, d'obtenir des statistiques sur la qualité globale des algorithmes. Pour cela, nous allons en fait calculer un pourcentage à l'erreur en prenant le rapport d'erreur, en le multipliant par 100 et en arrondissant la valeur obtenue à l'entier inférieur le plus proche : $\left\lfloor \frac{\mathbb{E}(A(I))-\beta(I)}{\omega(I)-\beta(I)} \cdot 100 \right\rfloor$. Dans la suite de ce chapitre nous appellerons cette mesure rapport à l'erreur.

5.2 Démarche expérimentale

Dans cette section, nous présentons notre démarche expérimentale, l'outil utilisé pour la réaliser et les échantillons de graphes sur lesquels nous avons effectué nos tests.

5.2.1 Méthode utilisée

Dès le début, et tout au long de cette thèse, nous avons développé un outil d'expérimentation pour le problème du vertex cover. Cet outil, réalisé en JAVA, n'a pas été conçu pour être diffusé et les programmes n'ont pas été optimisés « au mieux » car notre objectif a toujours été de comparer les algorithmes en fonction de la taille ou de la qualité des solutions retournées et non en fonction de leur temps d'exécution. Ainsi, nous ne donnerons ni les temps de calculs des divers algorithmes, ni de détails sur la machine sur laquelle nous avons mené nos expérimentations.

Le développement et l'utilisation de notre outil représente une part importante du travail effectué dans cette thèse, aussi bien en terme de temps qu'en terme d'idées. Par exemple, c'est après de nombreuses expérimentations que nous nous sommes rendus compte que ListRight est, en moyenne et pour tout graphe, $2-$approché.

La gestion de l'aléatoire. Chaque fois qu'un choix doit être effectué, il est fait de façon équiprobable. Effectuer des tirages aléatoires équiprobables revient à utiliser des générateurs aléatoires pseudo-uniformes, c'est-à-dire un générateur qui va tirer un nombre de fois équivalent chaque valeur possible lorsque le nombre de tirage est suffisamment élevé. La figure 5.1 est le résultat de 10000000 tirages aléatoires de valeurs comprises dans l'intervalle [0..9] en utilisant le générateur proposé par la classe Random de Java. Comme nous pouvons le constater, toutes les valeurs de l'intervalle ont été tirées un nombre de fois équivalent, ce qui montre que ce générateur est de bonne qualité. La même expérience a été menée pour les intervalles [0..1], [0..1000], et [0..100000] et a chaque fois le résultat à été le même, ce qui montre que la taille de l'intervalle n'a pas d'influence sur l'uniformité des tirages.

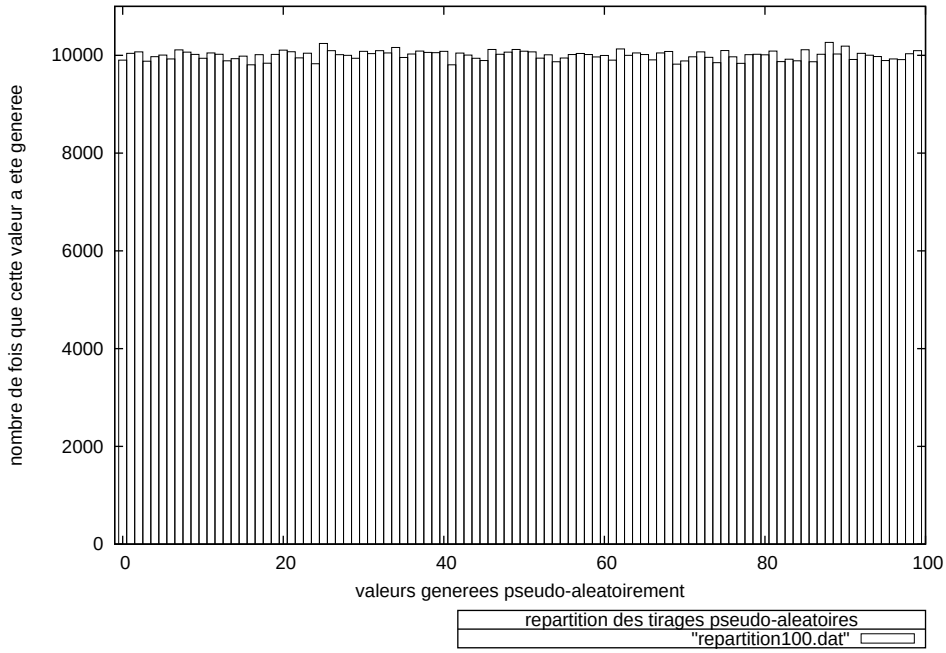


FIG. 5.1 – Répartition des valeurs pseudo-aléatoires obtenues après 10000000 de tirages dans [0 : 99].

Remarque. Les algorithmes SORTED LISTLEFT et SORTED LISTRIGHT nécessitent la génération de listes triées. Pour les obtenir, nous avions le choix entre deux méthodes : sélectionner une liste de façon uniforme parmi l'ensemble des listes possibles (soit $n!$, avec n la taille du graphe) et la trier ensuite, soit sélectionner de façon uniforme une liste triée parmi l'ensemble des listes triées. Nous avons choisi la première méthode pour sa simplicité d'une part, et pour placer les algorithmes SORTED LISTLEFT et SORTED LISTRIGHT sur un pied d'égalité avec les algorithmes LISTLEFT et LISTRIGHT. En effet, cela nous permet, étant donné une liste, de comparer les quatre algorithmes, ce qui est relativement intéressant.

Déroulement d'une expérience. Dans un premier temps, nous avons déterminé la liste des échantillons, c'est-à-dire les ensembles d'instances à tester au cours de notre expérience ainsi qu'une solution optimale* pour chaque graphe de chaque échantillon. Ensuite, pour chaque échantillon, nous avons effectué 10000 itérations de chaque algorithme sur chaque instance de l'échantillon.

5.2.2 Nos échantillons de tests

Calculer une solution optimale pour un graphe de taille même modeste peut être relativement long, surtout lorsqu'il n'existe pas de structure exploitable dans le graphe. Pour cette raison, nous nous sommes restreints à des tailles de graphes modestes. Cependant, si on considère juste les performances relatives en terme de taille des solutions des algorithmes, nous pouvons affirmer que les résultats obtenus dans cette étude sont valables pour des tailles de graphes plus importantes.

*Pour calculer une solution optimale pour chaque instance, nous avons utilisé la programmation linéaire en nombre entiers. Pour chaque graphe, nous avons généré un fichier de contraintes et nous avons utilisé le programme $LPSOLVE$ pour trouver une solution optimale.

Cette affirmation provient de notre expérience, obtenue tout au long de cette thèse et des diverses expériences menées. Dans un souci de concision, ces résultats ne seront pas donnés ici, mais feront éventuellement l'objet d'une prochaine publication.

Les graphes de Erdös-Renyi. Nous avons généré 4500 graphes aléatoires de Erdös-Renyi. Plus précisément, pour chaque taille n avec $n \in \{20, 40, 60, 80, 100\}$ et pour chaque probabilité d'arête p avec $p \in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$, nous avons généré 100 graphes $G_{n,p}$. Dans le chapitre 3, nous avons donné deux formules permettant de calculer l'espérance exacte des algorithmes ListLeft et ListRight sur les graphes de Erdös-Renyi et nous avons obtenu une figure représentant l'évolution de leur espérance pour $n = 100$ lorsque l'on fait varier la probabilité d'existence d'une arête. Pour chaque graphe de notre échantillon, nous avons calculé l'espérance expérimentale des algorithmes ListLeft et ListRight, puis, pour chaque ensemble de 100 graphes ayant le même nombre de sommets et la même probabilité d'arête, nous avons fait la moyenne des espérances obtenues. La figure 5.2 montre que la moyenne obtenue pour $n = 100$ est en adéquation parfaite avec les prévisions théoriques du chapitre 3. Nous avons obtenu les mêmes résultats pour les différentes valeurs de notre échantillon. Ce résultat nous indique que notre échantillon de graphes aléatoires est représentatif de l'ensemble des graphes aléatoires. Il n'est donc pas nécessaire d'effectuer nos expérimentation sur un éventail plus large de graphes aléatoires. De plus, le fait que nos algorithmes se comportent conformément aux prévisions nous prouve que le nombre d'itérations que nous avons effectué est suffisamment important. On peut donc légitimement penser qu'il en sera de même pour les autres algorithmes.

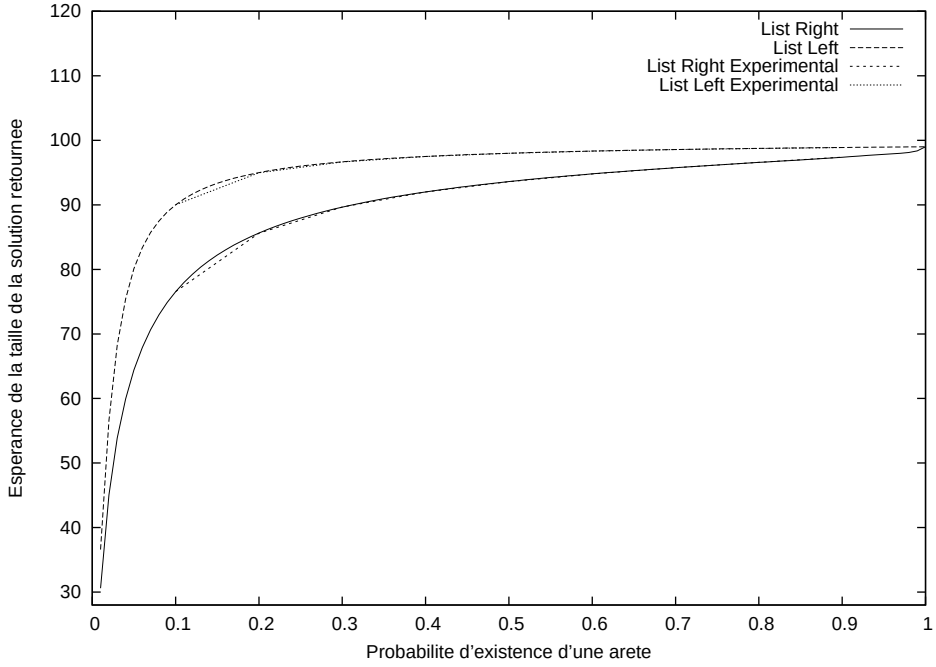


Fig. 5.2 – Comparaison entre l'espérance analytique obtenue dans le chapitre 3 et l'espérance expérimentale obtenue sur notre échantillon de graphes aléatoires pour les algorithmes ListLeft et ListRight

Benchmarks with Hidden Optimum Solutions (BHOSLIB). Pour évaluer les performances d'un algorithme, il est très fréquent d'utiliser des benchmarks, c'est-à-dire un ensemble d'instances qui va servir de base commune à l'évaluation des différents algorithmes. Nous avons choisis d'utiliser les benchmarks proposées sur le site « http ://www.nlsde.buaa.edu.cn/˜kexu/benchmarks/graph-benchmarks.htm », car toutes les instances ont été générées à partir du Modèle RB (voir [XL06]) qui est reconnu pour générer des instances difficiles à résoudre. La génération de ces instances se fait de la façon suivante :

1. Générer n cliques disjointes, chacune avec n^α sommets (avec $\alpha > 0$ une constante) ;
2. Sélectionner aléatoirement 2 cliques différentes et créer sans répétition $pn^{2\alpha}$ arêtes aléatoires entre ces deux cliques (avec $0 < p < 1$ une constante) ;
3. Recommencer la deuxième étape $rn \ln n - 1$ fois (avec $r > 0$ une constante).

Dans [XBHL07], les auteurs expliquent comment déterminer de bonnes valeurs pour les constantes p, r et α. Les benchmarks obtenus à partir de cette méthode sont les suivants[†] :

– 5 instances de 450 sommets (30 cliques) avec une solution optimale de 420 sommets.
– 5 instances de 595 sommets (35 cliques) avec une solution optimale de 560 sommets.
– 5 instances de 760 sommets (40 cliques) avec une solution optimale de 720 sommets.
– 5 instances de 945 sommets (45 cliques) avec une solution optimale de 900 sommets.
– 5 instances de 1150 sommets (50 cliques) avec une solution optimale de 1100 sommets.
– 5 instances de 1272 sommets (53 cliques) avec une solution optimale de 1219 sommets.
– 5 instances de 1400 sommets (56 cliques) avec une solution optimale de 1344 sommets.
– 5 instances de 1534 sommets (59 cliques) avec une solution optimale de 1475 sommets.

Les arbres. Nous avons comparé nos algorithmes sur des arbres. Ces graphes connexes ont une très faible densité et il est très facile de retourner une solution optimale en temps polynomial (c'est le cas de l'algorithme Greedy Independent Cover par exemple). Il existe de nombreuses méthodes pour générer un arbre. La méthode de génération utilisée pour obtenir notre échantillon est la suivante :

1. On débute avec un graphe complet à $n > 3$ sommets ;
2. Retirer une arête au hasard de sorte que le graphe résultant soit connexe ;
3. Tant que le nombre d'arêtes est supérieur à $n - 1$ on recommence l'étape 2.

En utilisant cette méthode, et pour $n = 50$, 100, 500 et 1000, nous avons généré 1000 arbres, soit 4000 arbres au total.

Graphes de pires cas. Lorsque cela était possible, nous avons généré des instances qui piègent un algorithme en moyenne et en terme de rapport à l'erreur. L'intérêt de cet échantillon est double. D'une part, il est nécessaire de montrer que les différents algorithmes peuvent être mauvais en moyenne sur au moins une classe de graphes, et il est intéressant, d'autre part, d'observer le comportement des autres algorithmes sur cette même instance.

[†]Pour chaque instance, la taille du minimum vertex cover est connue, mais la solution en elle même n'est pas fournie.

– Pour l'algorithme Sorted ListLeft, nous avons utilisé les graphes $ProW$ (voir chapitre 2) qui forcent toute exécution de l'algorithme à retourner une solution de taille $\frac{N+1}{2}|OPT|$, avec N la dimension du graphe, une constante aussi grande que l'on veut.
– Pour l'algorithme Sorted ListRight, une simple modification des graphes $ProW$ permet d'obtenir exactement le même comportement. Il suffit de relier chaque sommet de F_1 à chaque sommet de V_2 (voir la figure 5.3 pour $N = 3$ par exemple). Cette modification fait que toute liste $\mathcal{L}$ triée par degrés décroissants contient en premier les sommets de V_2, suivis par les sommets de U, puis ceux de F_1, puis les sommets de V_1 et enfin les sommets de F_2. En reprenant les différents arguments présentés dans la preuve du théorème 12, on obtient que toute exécution de Sorted ListRight va retourner une solution de taille $\frac{N+1}{2}|OPT|$, avec N la dimension du graphe.

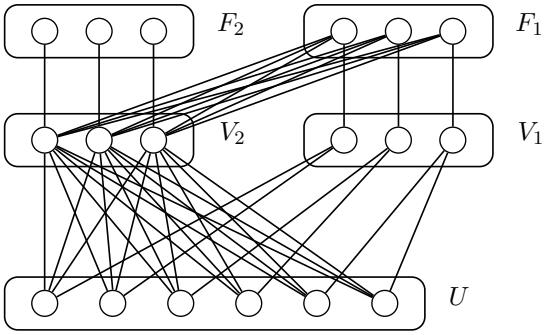


Fig. 5.3 – Exemple de graphe $ProW$ modifié pour $N = 3$.

– Pour l'algorithme Edge Deletion, il suffit de considérer des graphes bipartis complets. Ces graphes contiennent un couplage parfait et toute exécution de Edge Deletion va obligatoirement retourner tous les sommets.
– Pour l'algorithme Maximum Degree Greedy, nous avons utilisé les graphes $Anti - MDG$ modifiés (c'est-à-dire sans les sommets de degré 2 dans l'ensemble C) présentés à la fin du chapitre 1.
– Pour l'algorithme Greedy Independent Cover, nous avons utilisé les graphes présentés par Avis et Imamura dans [AI07] pour montrer que tout algorithme de liste triée est au plus $\frac{\sqrt{\Delta}}{2}-$approché.
– Pour l'algorithme ListLeft, il n'a pas été nécessaire de déterminer une classe particulière pour piéger l'algorithme car il possède de mauvaises performances en moyenne sur la plupart des instances de l'échantillon.
– Trouver une classe de graphes qui piègent un algorithme en moyenne n'est pas une tâche simple. Nous avons vu que l'algorithme ListRight est, pour tout graphe, au plus 2−approché en moyenne et que ce rapport d'approximation moyen tend à être atteint pour les étoiles lorsque le nombre de feuilles tend vers l'infini. Cependant, le rapport à l'erreur de ListRight sur les étoiles est très proche de 0 car ses performances moyennes sont très éloignées du pire cas (toutes les feuilles plus le centre de l'étoile). Nous n'avons pas réussi à trouver une seule instance qui parvienne à piéger ListRight en terme de rapport à l'erreur. Pour cette raison, nous n'évaluerons ses performances que sur les instances qui piègent les autres algorithmes.

Au final, notre échantillon contient :

- 5 graphes Anti-MDG modifiés de dimension 30 à 34 ;
- 5 graphes de Avis et Imamura de dimension 30 à 34 ;
- 5 graphes $ProW$ de dimension 30 à 34 ;
- 5 graphes $ProwW$ modifiés de dimension 30 à 34 ;
- 5 graphes bipartis complets dont le nombre de sommets est de 2×10, 2×15, 2×20, 2×25 et 2×30.

Graphes admettant une régularité structurelle. Nous avons aussi souhaité comparer les algorithmes sur des graphes usuels possédant une structure régulière tels que les grilles, les tores, les hypercubes et les graphes réguliers. Globalement, ces graphes admettent une solution optimale qui contient la moitié des sommets (à un sommet près pour les grilles et les tores). Tous les algorithmes se trouvent donc sur un pied d'égalité puisqu'ils sont tous au plus 2−approchés. Plus précisément, cet échantillon contient :

- toutes les grilles et tous les tores de dimensions 2×2 à 20×20 ;
- les hypercubes de dimension 3 à 12 ;
- 36 graphes réguliers dont le nombre de sommets est compris entre 28 et 98. Le degré de ces graphes varie de 3 à 8.

5.3 Les résultats

Les résultats sont regroupés par échantillon, à raison d'un échantillon par page. Chaque page contient 8 figures qui représentent les performances de chaque algorithme. L'abscisse d'une figure représente le pourcentage d'erreur obtenu par l'algorithme pour une instance de l'échantillon et l'ordonnée représente le nombre d'instances de l'échantillon ayant obtenu ce rapport à l'erreur.

Dans la suite, nous dirons qu'un algorithme A domine un algorithme B sur une instance I lorsque le rapport à l'erreur de l'algorithme A sur l'instance I est inférieur à celui de l'algorithme B. Il faut bien garder en mémoire qu'une telle relation de dominance ne signifie pas que toute exécution de l'algorithme A est meilleure que celles de B, mais qu'il s'agit d'un résultat obtenu en moyenne après 10000 itérations de chaque algorithme, ce qui donne la tendance générale des solutions retournées.

Les figures 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10 et 5.11 sont obtenues en appliquant les algorithmes sur l'échantillon de graphes aléatoires.

Les figures 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18 et 5.19 sont obtenues en appliquant les algorithmes sur l'échantillon des arbres.

Les figures 5.20, 5.21, 5.22, 5.23, 5.24, 5.25, 5.26 et 5.27 sont obtenues en appliquant les algorithmes sur l'échantillon de benchmarks BHOSLIB.

Les figures 5.28, 5.29, 5.30, 5.31, 5.32, 5.33, 5.34 et 5.35 sont obtenues en appliquant les algorithmes sur l'échantillon des graphes de pire cas moyen.

Échantillon	GIC	MDG	SLL	SLR	LL	LR	ED	DFS
Erdös-Renyi	0-25	1-33	25-88	0-43	63-82	13-42	68-97	36-91
Arbres	0-0	0-9	2-19	0-6	29-42	7-17	46-67	18-48
BHOSLIB	15-20	30-44	74-90	24-36	81-85	32-34	85-88	89-98
Graphes réguliers	0-8	0-32	0-83	0-55	16-83	0-55	68-100	50-99

TAB. 5.1 – Tableau récapitulatif des intervalles de pourcentages d'erreur obtenus par chaque algorithme pour chaque échantillon. Le premier nombre représente le meilleur pourcentage d'erreur obtenu sur l'échantillon et le deuxième représente le plus mauvais.

Les figures 5.36, 5.37, 5.38, 5.39, 5.40, 5.41, 5.42 et 5.43 sont obtenues en appliquant les algorithmes sur l'échantillon de graphes réguliers.

Nous allons tout d'abord considérer l'échantillon des graphes de pire cas. L'algorithme GREEDY INDEPENDENT COVER est optimal sur toutes les instances excepté sur les 5 spécialement conçues pour le piéger pour lesquels il obtient un rapport à l'erreur proche de 95%. L'algorithme MAXIMUM DEGREE GREEDY est lui aussi très bon puisqu'il est optimal sur la majorité des instances et que son rapport à l'erreur est proche des 70% sur les 5 graphes $Anti-MDG$ modifiés. Notons toutefois que contrairement à GREEDY INDEPENDENT COVER il possède un rapport à l'erreur de 18% sur 5 autres instances.
L'algorithme LISTLEFT est relativement mauvais puisque son rapport à l'erreur sur toutes les instances sauf 5 est supérieur à 70%, et même superieur à 90% pour 18 d'entre elles. LISTRIGHT se comporte étonnamment bien. En effet, son rapport à l'erreur ne dépasse jamais les 20% et il est même optimal sur les graphes bipartis complets. Il convient cependant de modérer ce résultat par le fait que l'échantillon ne contenait pas de graphe spécialement conçu pour le piéger en moyenne.
L'algorithme SORTED LISTLEFT semble légèrement meilleur que l'algorithme LISTLEFT sauf pour 1 instance dont le rapport à l'erreur est de 99%. Cependant, même si ses performances sont meilleures que celles de LISTLEFT, l'algorithme SORTED LISTLEFT n'a jamais de rapport à l'erreur de moins de 10%. Au contraire, l'algorithme SORTED LISTRIGHT est optimal sur la majorité des instances, sauf sur 5 pour lesquelles il obtient un rapport à l'erreur de 16% et les 5 conçues pour le piéger et sur lesquelles il obtient un rapport à l'erreur proche de 95%.
Les deux algorithmes 2−approchés, à savoir DEPTH FIRST SEARCH et EDGE DELETION obtiennent des performances similaires, même si EDGE DELETION est légèrement plus mauvais. Ils n'obtiennent jamais de rapport à l'erreur meilleur que 8% et obtiennent un très mauvais score (plus de 98%) pour les instances conçues pour les piéger.
Ces résultats soulignent le mauvais comportement des algorithmes lorsqu'ils sont confrontés à des instances conçues pour les piéger en moyenne et en terme de rapport à l'erreur. Il convient de noter que les algorithmes GREEDY INDEPENDENT COVER, MAXIMUM DEGREE GREEDY, SORTED LISTRIGHT et LISTRIGHT obtiennent des rapports à l'erreur de 0% sur de nombreuses instances. Cela signifie que toutes les exécutions des algorithmes sur ces instances ont retourné une solution optimale.

Nous allons maintenant nous concentrer sur les autres échantillons. Le tableau 5.1 récapitule pour chaque algorithme et chaque échantillon le meilleur rapport à l'erreur obtenu ainsi que le plus mauvais.

Échantillon	1	2	3	4	5	6	7	8
Erdös-Renyi	GIC	MDG	SLR	LR	SLL	LL	DFS	ED
Arbres	GIC	SLR	MDG	SLL	LR	DFS	LL	ED
BHOSLIB	GIC	SLR	LR	MDG	LL	SLL	ED	DFS
Graphes réguliers	GIC	MDG	SLR	LR	SLL	LL	DFS	ED
Graphes de pire cas	GIC	MDG	SLR	LR	SLL	LL	DFS	ED

Tab. 5.2 – Tableau des classements des algorithmes par échantillon

Le résultat le plus remarquable est le très bon comportement de l'algorithme Greedy Independent Cover. Le pourcentage d'erreur de cet algorithme est meilleur que celui de tous les autres algorithmes, et ce pour toutes les instances de nos échantillons. Cet algorithme est même optimal en moyenne pour de très nombreuses instances, comme sur les graphes de Erdös-Renyi ayant une faible probabilité d'arête, les arbres et une grande partie des graphes réguliers. Son pourcentage d'erreur n'excède jamais les 25%, ce qui en fait un excellent algorithme.

Au contraire, le comportement de l'algorithme Edge Deletion est remarquablement mauvais compte tenu de son rapport d'approximation. Cet algorithme est même dominé par tous les autres algorithmes sur les échantillons des graphes de Erdös-Renyi et les benchmarks BHOSLIB, et par tous les algorithmes excepté Depth First Search pour les graphes réguliers et par les 4 algorithmes Greedy Independent Cover, Maximum Degree Greedy, Sorted ListRight et ListRight sur les arbres. Son pourcentage d'erreur n'est jamais inférieur à 46%, ce qui montre que cet algorithme ne retourne que des solutions de mauvaise qualité.

Tous les algorithmes testés ont visiblement des difficultés avec les instances qui sont difficiles à résoudre car les pourcentages d'erreur de tous les algorithmes sont plus élevés sur les graphes de l'échantillon BHOSLIB que sur les autres échantillons, et aucun ne fait mieux que 15%.

Trier les listes par degrés décroissants semble être une bonne idée puisque cela permet d'obtenir, globalement, de meilleurs résultats. Cependant, le tri à tout de même détérioré la qualité des solutions pour plusieurs instances de l'échantillon des graphes aléatoires et de l'échantillon BHOSLIB.

En observant les résultats du tableau 5.1, on peut remarquer un détail intéressant : le plus mauvais pourcentage d'erreur obtenu par l'algorithme ListRight est toujours meilleur que le meilleur pourcentage d'erreur obtenu par Edge Deletion, ce qui semble indiquer que l'espérance de la taille des solutions retournées par ListRight est toujours inférieure à l'espérance de la taille des solutions retournées par Edge Deletion. Nous avons tenté de construire un graphe qui permette de contredire cette hypothèse sans y parvenir et nous avons lancé des milliers de tests sur des graphes aléatoires. Le mieux que nous ayons trouvé est un graphe pour lequel les deux algorithmes ont la même espérance. Ces résultats nous conduisent à poser la conjecture suivante :

Conjecture : Pour tout graphe G, on a : $\mathbb{E}(\text{ListRight}(G)) \leq \mathbb{E}(\text{Edge Deletion}(G))$.

À partir de tous ces résultats, nous pouvons effectuer un classement des algorithmes, échantillon par échantillon. Ce classement, présenté dans le tableau 5.2, est obtenu en fonction des intervalles de pourcentages d'erreur et de la répartition des pourcentages d'erreur obtenus.

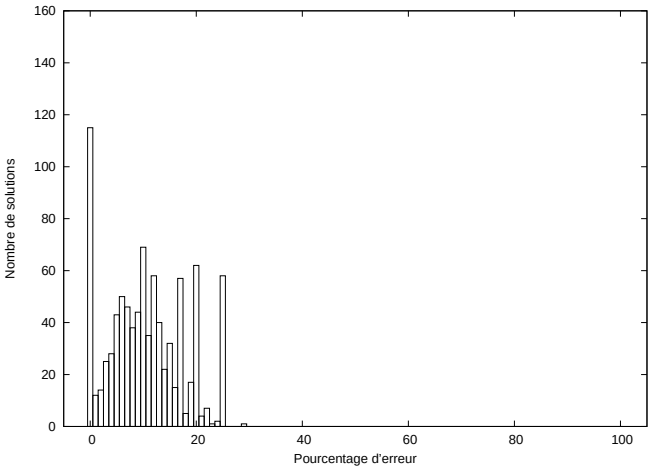


FIG. 5.4 – GREEDY INDEPENDENT COVER

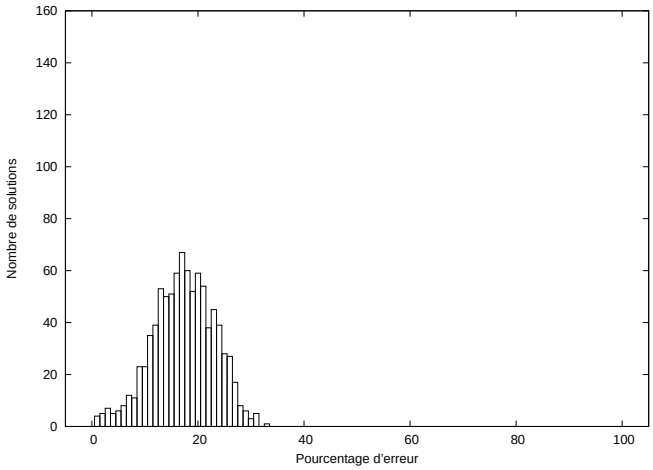


FIG. 5.5 – MAXIMUM DEGREE GREEDY

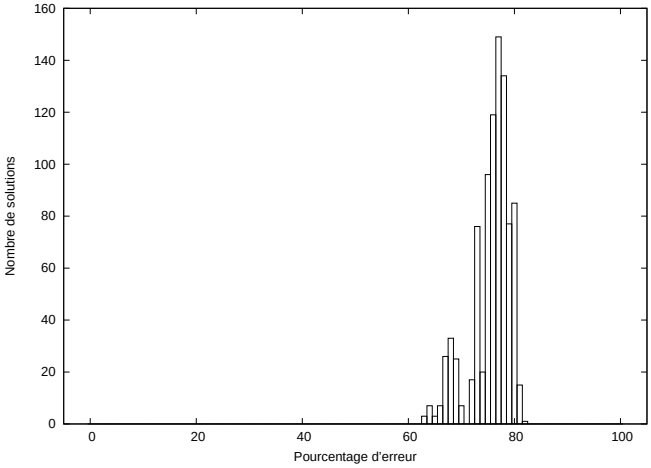


FIG. 5.6 – LISTLEFT

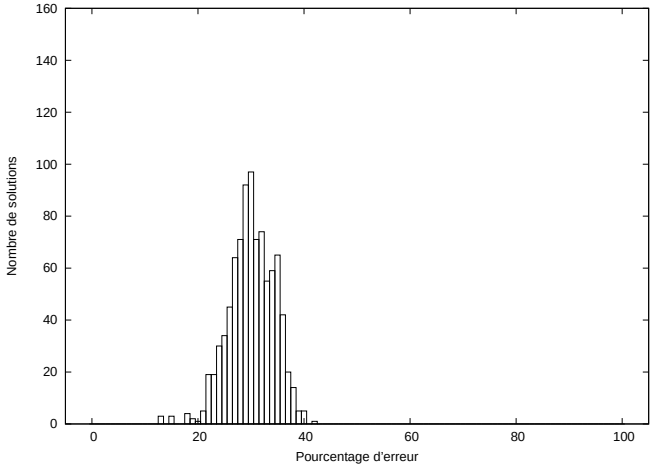


FIG. 5.7 – LISTRIGHT

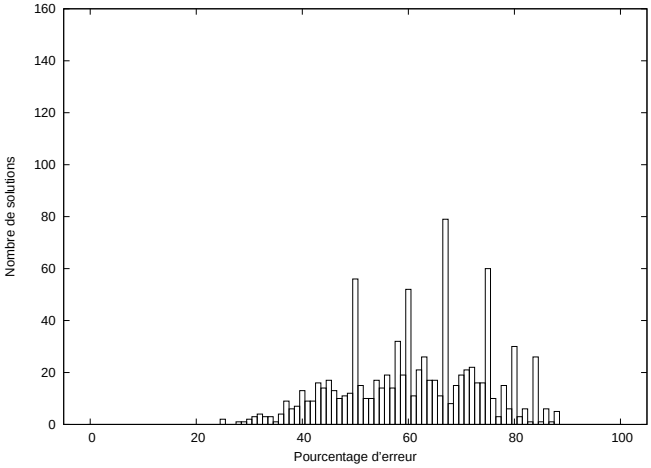


FIG. 5.8 – SORTED LISTLEFT

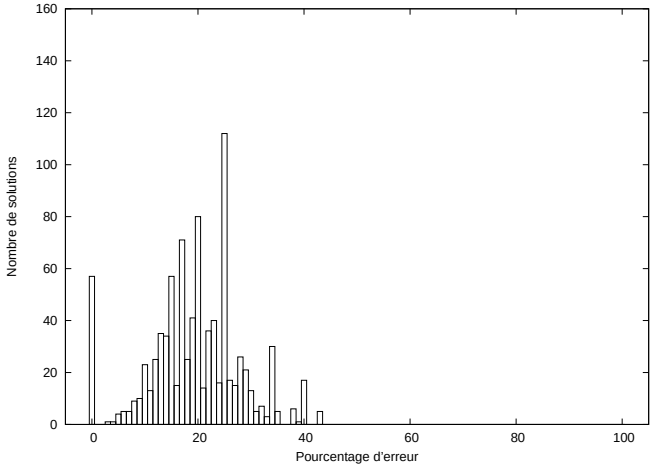


FIG. 5.9 – SORTED LISTRIGHT

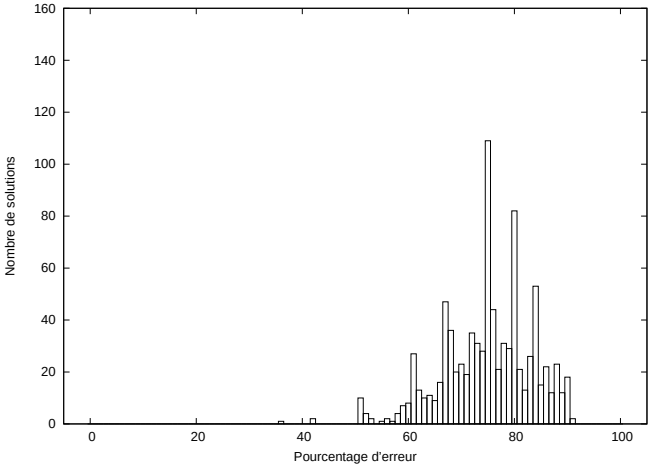


FIG. 5.10 – DEPTH FIRST SEARCH

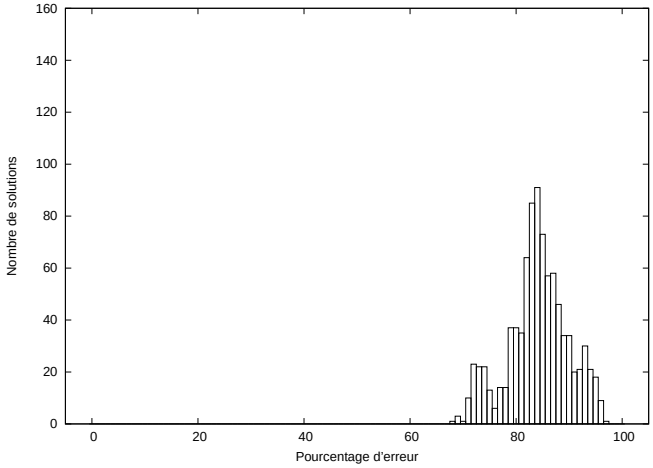


FIG. 5.11 – EDGE DELETION

Échantillon des graphes aléatoires

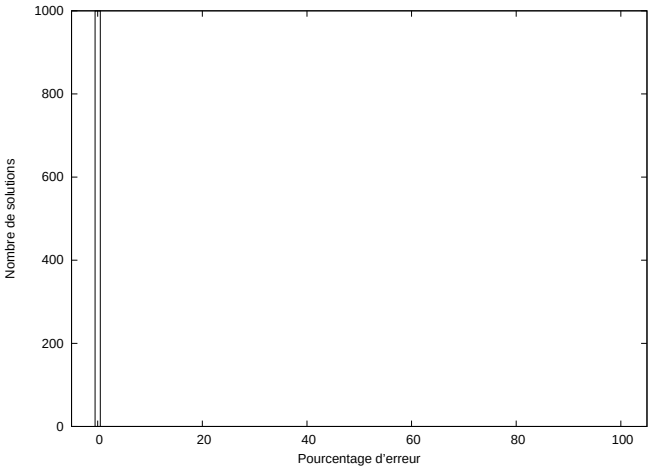


FIG. 5.12 – GREEDY INDEPENDENT COVER

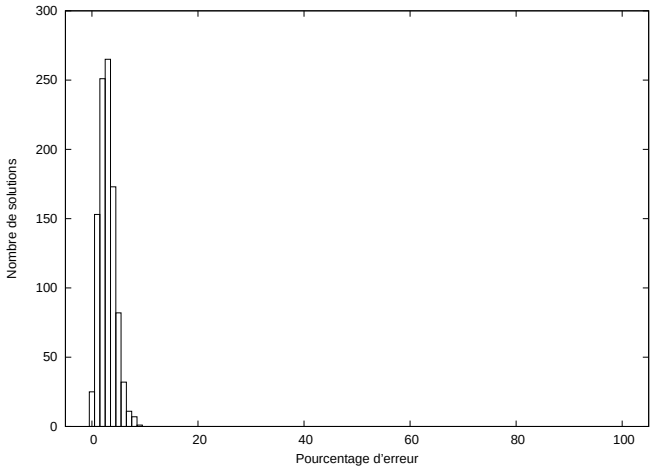


FIG. 5.13 – MAXIMUM DEGREE GREEDY

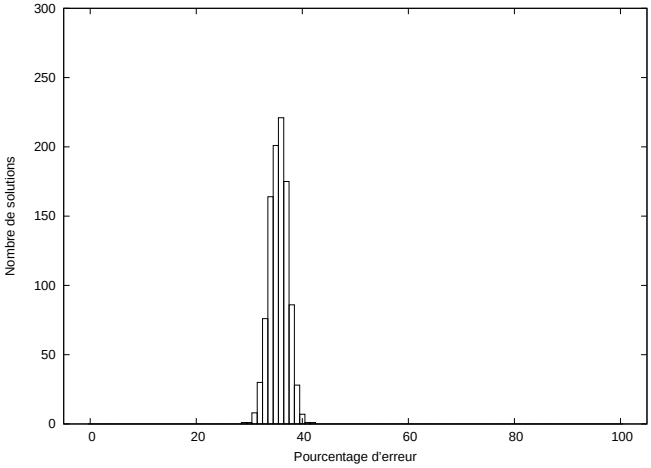


FIG. 5.14 – LISTLEFT

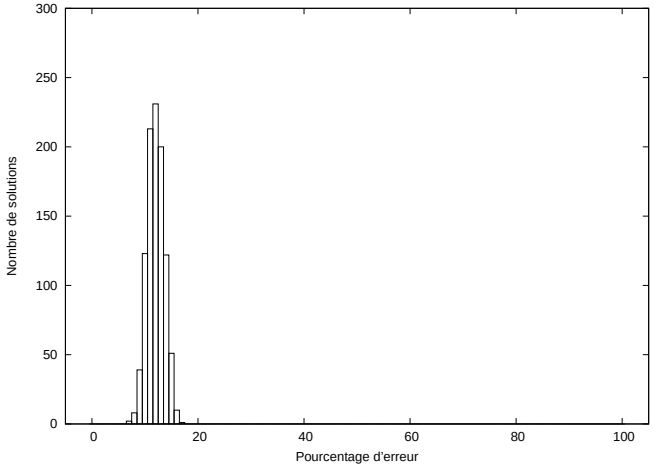


FIG. 5.15 – LISTRIGHT

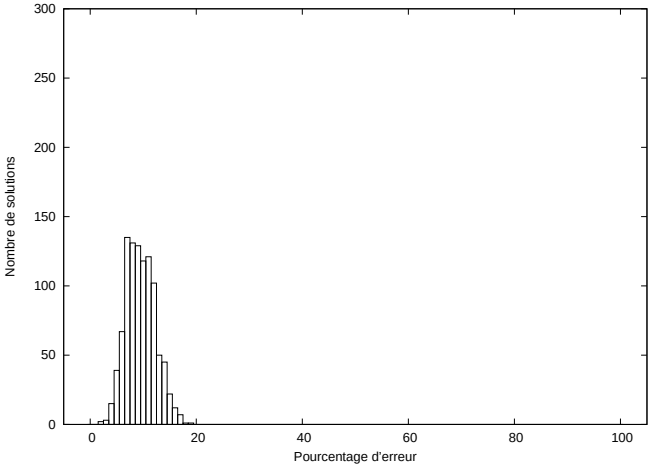


FIG. 5.16 – SORTED LISTLEFT

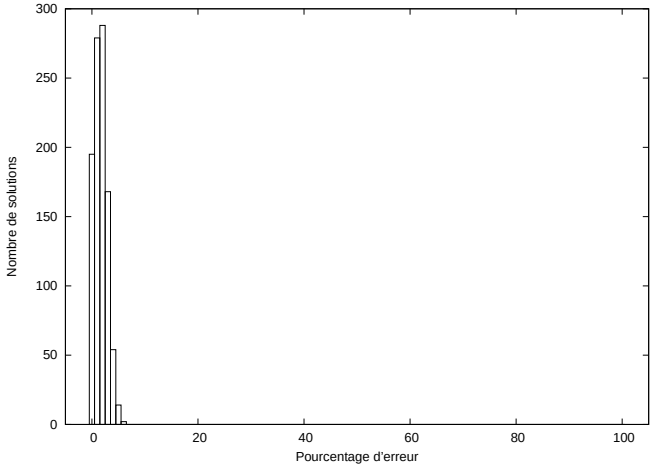


FIG. 5.17 – SORTED LISTRIGHT

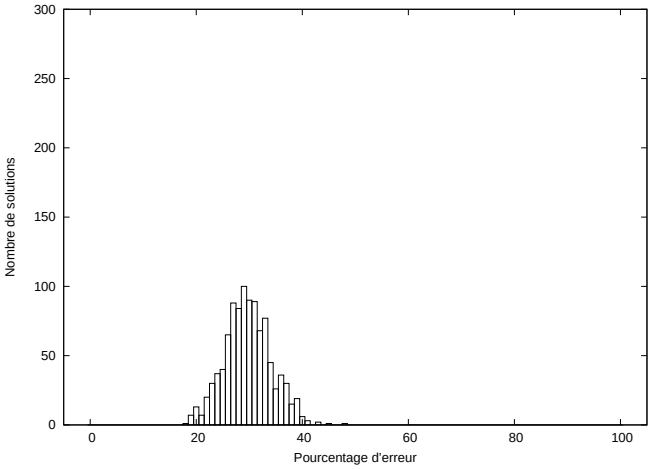


FIG. 5.18 – DEPTH FIRST SEARCH

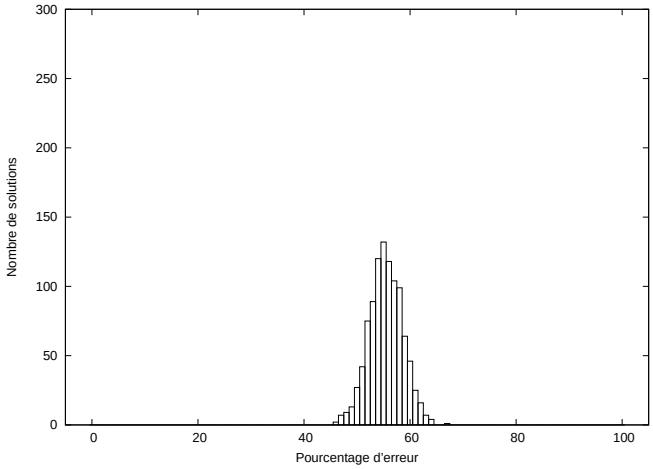


FIG. 5.19 – EDGE DELETION

Échantillon des arbres

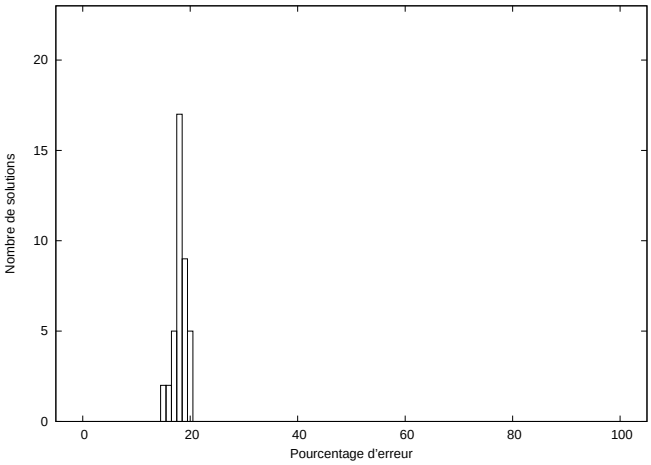


FIG. 5.20 – GREEDY INDEPENDENT COVER

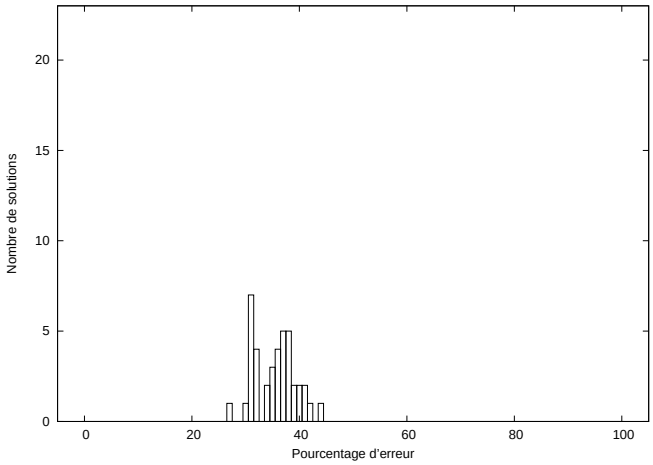


FIG. 5.21 – MAXIMUM DEGREE GREEDY

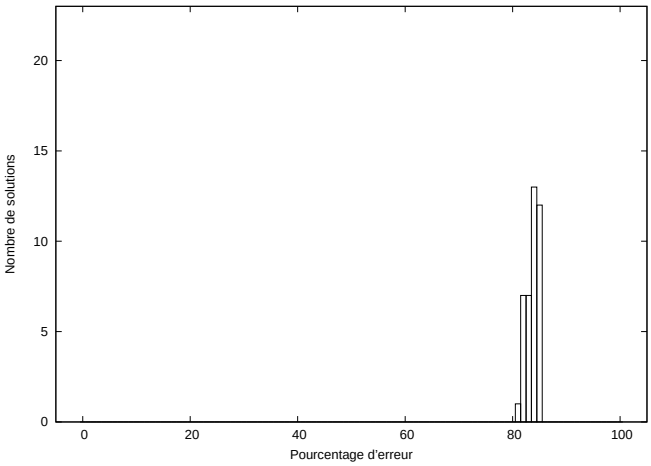


FIG. 5.22 – LISTLEFT

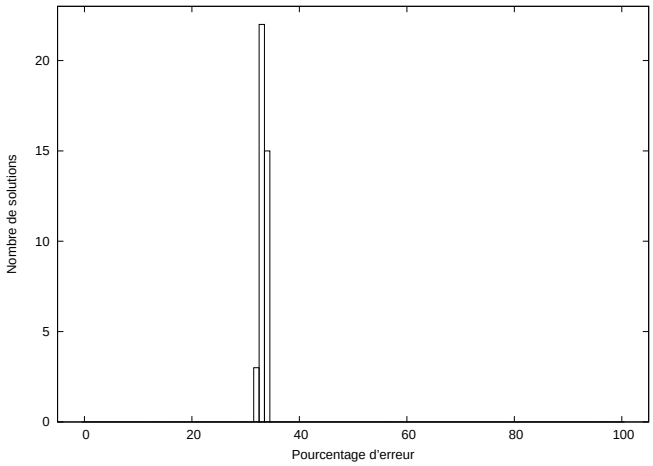


FIG. 5.23 – LISTRIGHT

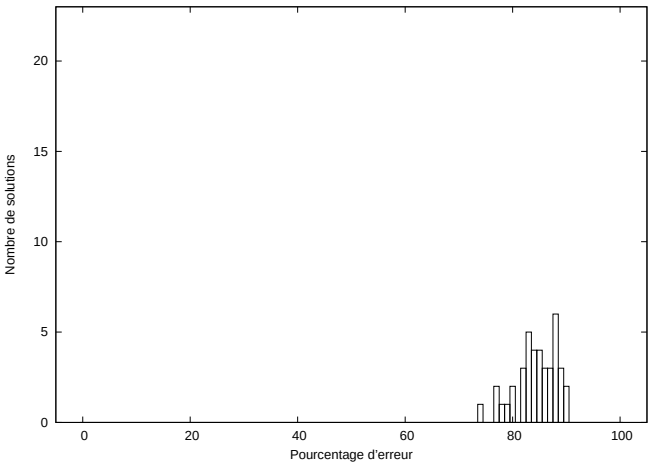


FIG. 5.24 – SORTED LISTLEFT

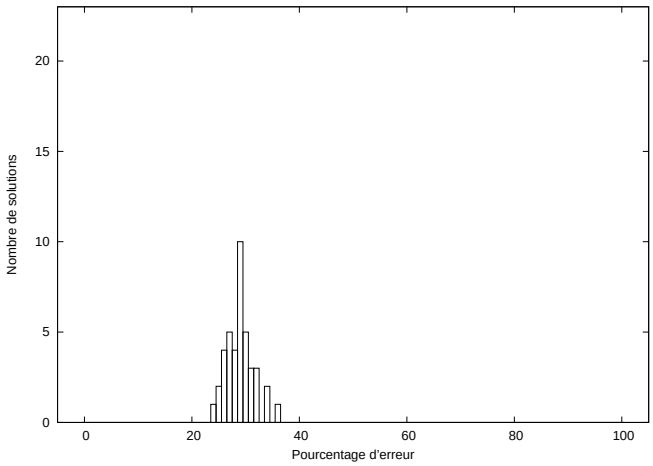


FIG. 5.25 – SORTED LISTRIGHT

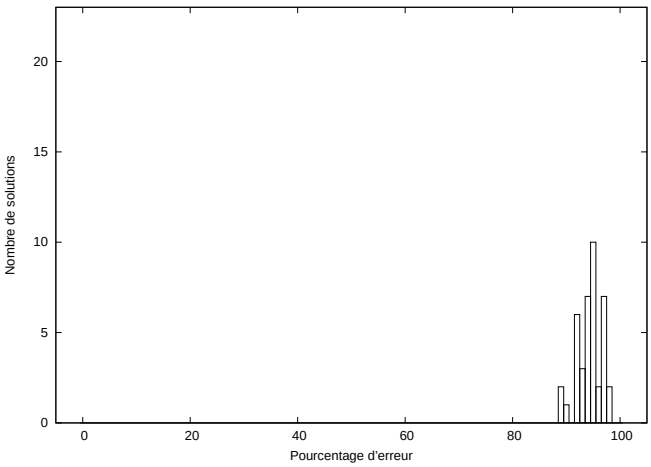


FIG. 5.26 – DEPTH FIRST SEARCH

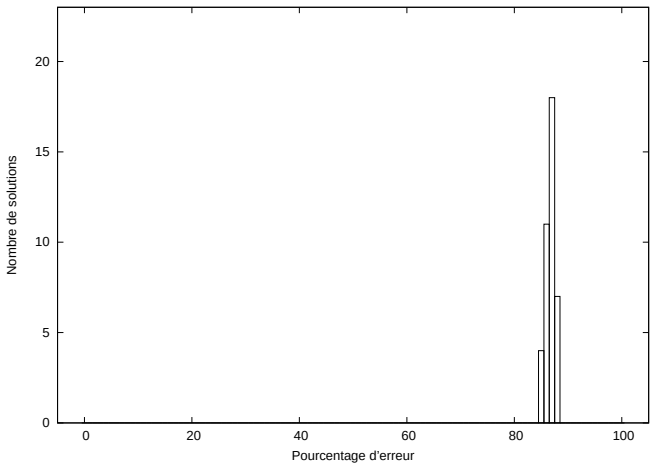


FIG. 5.27 – EDGE DELETION

Échantillon de benchmarks BHOSLIB

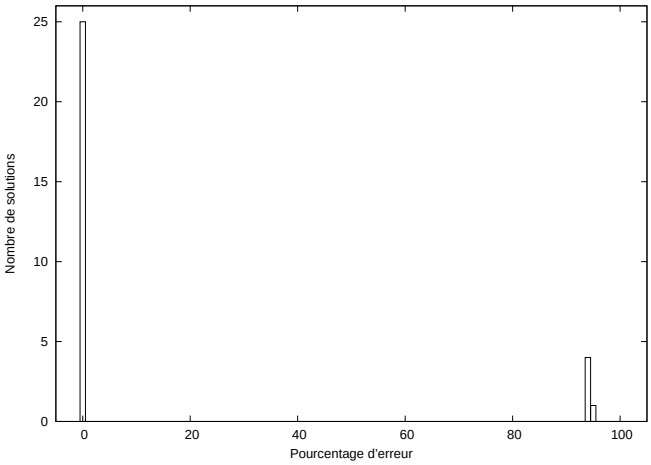


FIG. 5.28 – GREEDY INDEPENDENT COVER

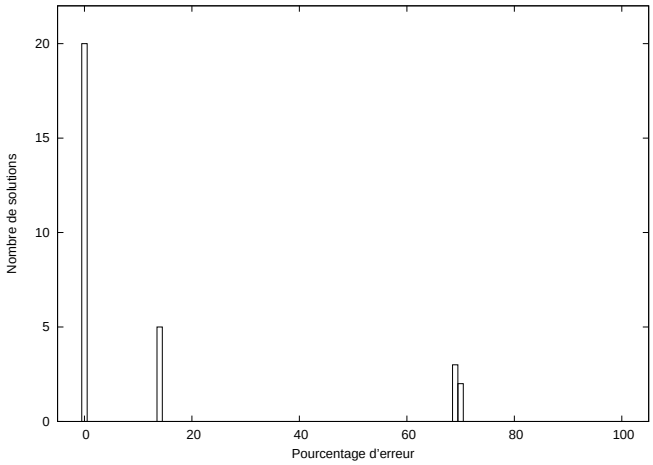


FIG. 5.29 – MAXIMUM DEGREE GREEDY

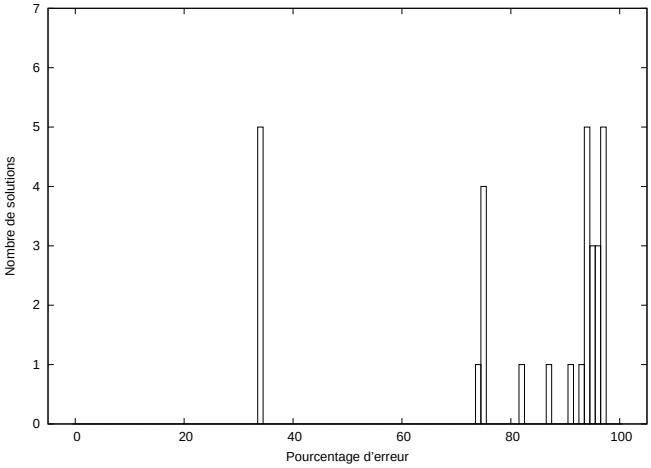


FIG. 5.30 – LISTLEFT

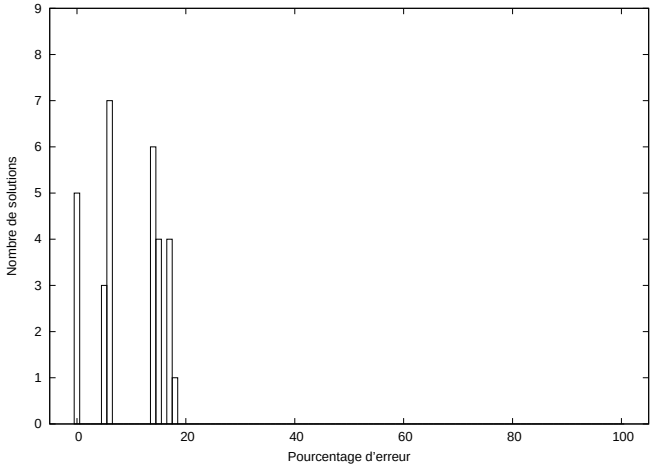


FIG. 5.31 – LISTRIGHT

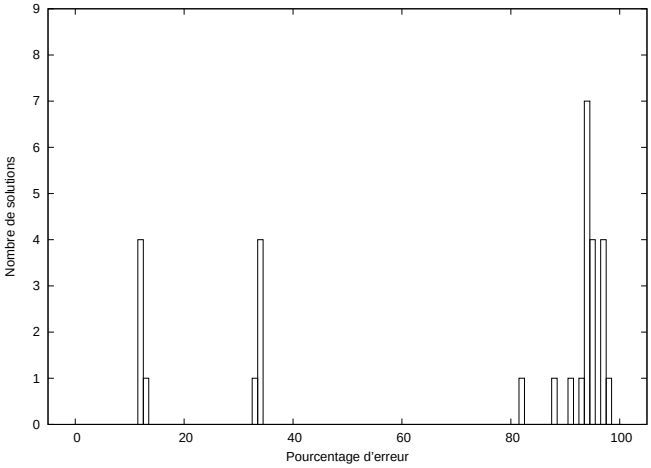


FIG. 5.32 – SORTED LISTLEFT

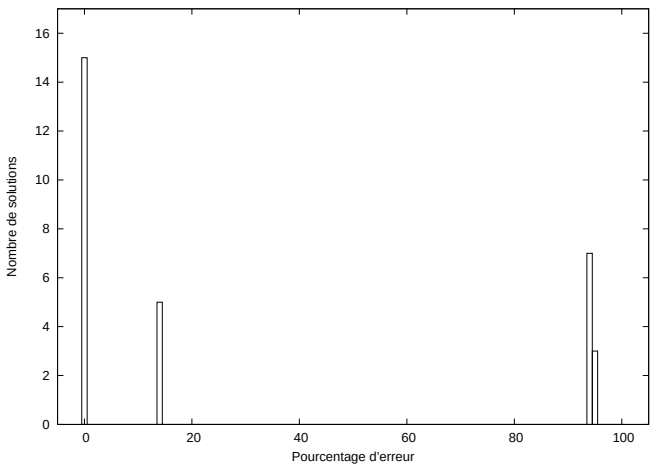


FIG. 5.33 – SORTED LISTRIGHT

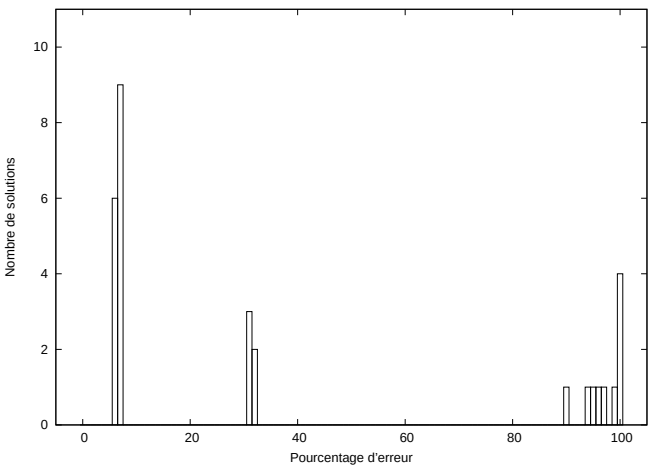


FIG. 5.34 – DEPTH FIRST SEARCH

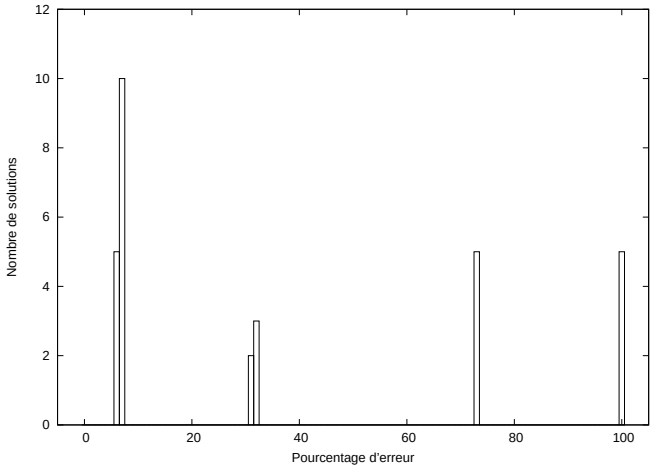


FIG. 5.35 – EDGE DELETION

Échantillon des graphes de pire cas moyen

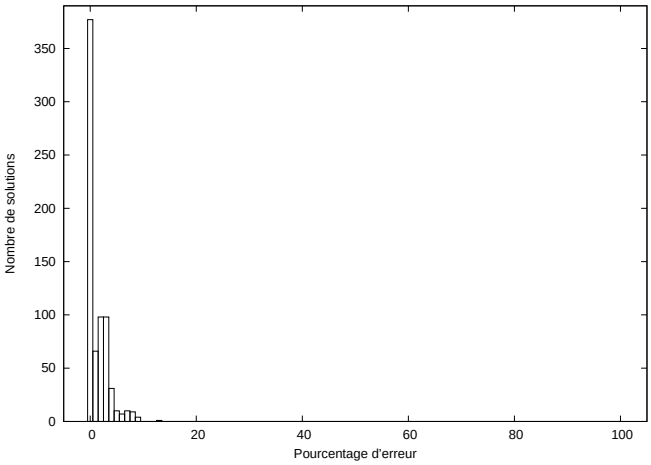


FIG. 5.36 – GREEDY INDEPENDENT COVER

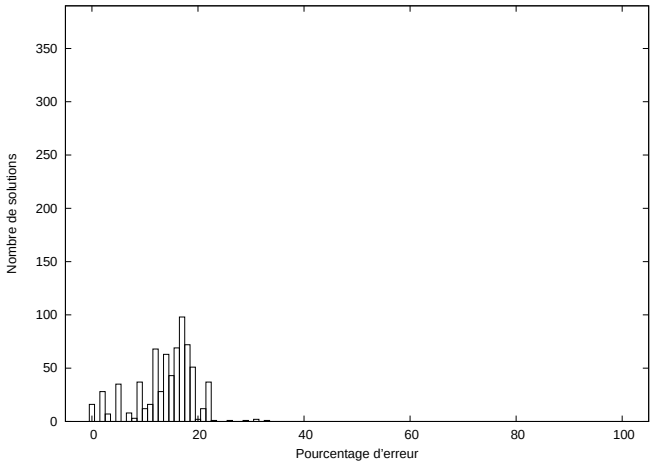


FIG. 5.37 – MAXIMUM DEGREE GREEDY

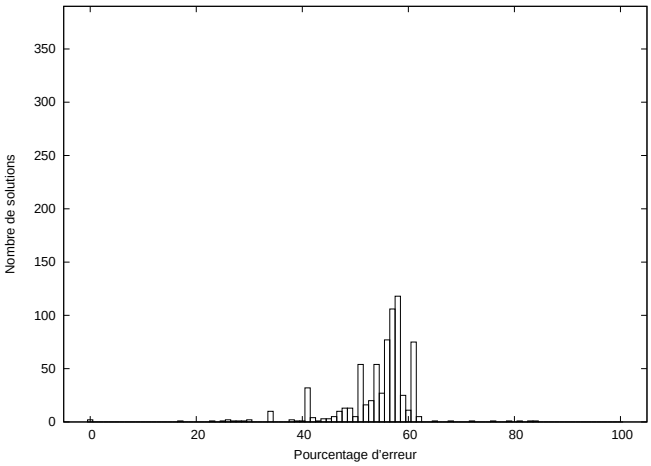


FIG. 5.38 – LISTLEFT

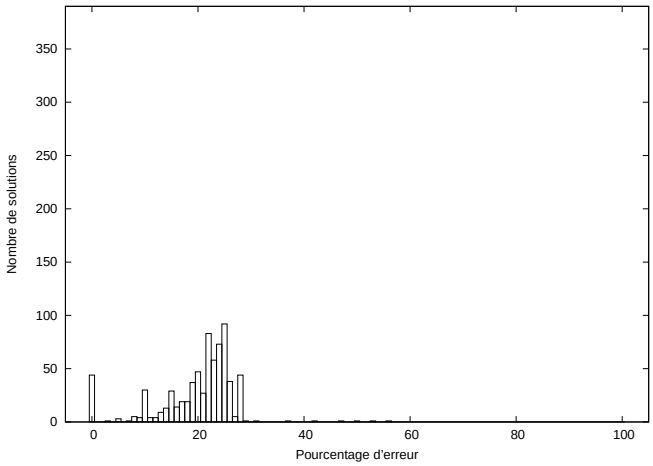


FIG. 5.39 – LISTRIGHT

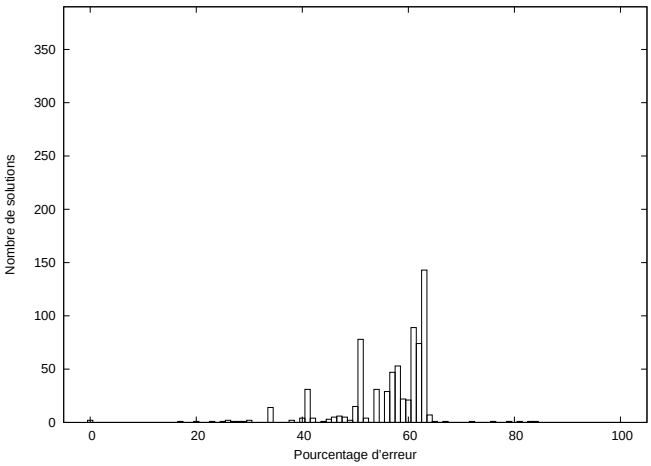


FIG. 5.40 – SORTED LISTLEFT

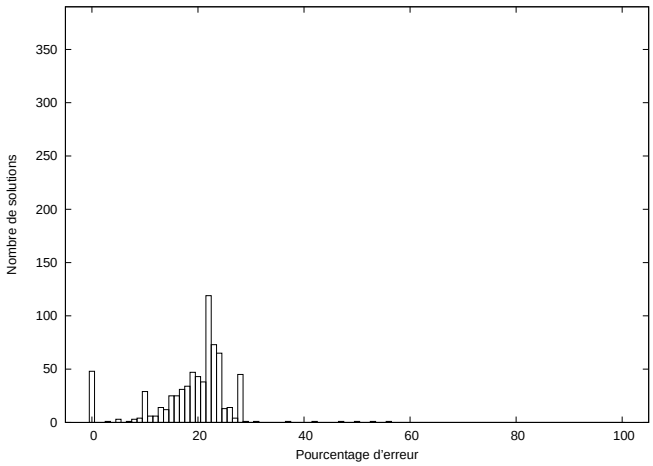


FIG. 5.41 – SORTED LISTRIGHT

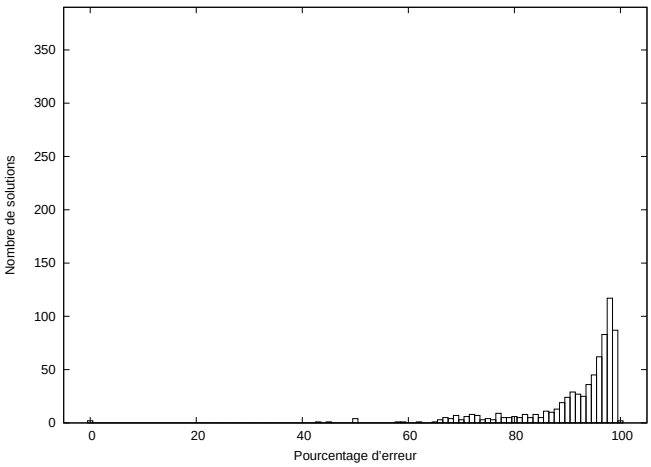


FIG. 5.42 – DEPTH FIRST SEARCH

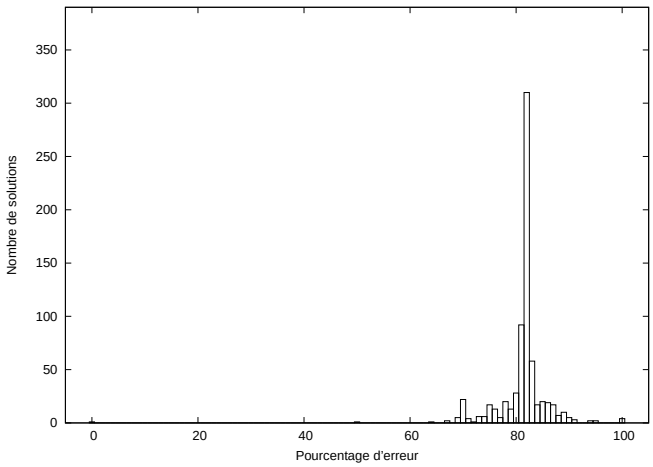


FIG. 5.43 – EDGE DELETION

Échantillon des graphes réguliers

5.4 Bilan

Dans ce chapitre, nous avons mené une comparaison expérimentale de la qualité des solutions retournées par les algorithmes que nous avons déjà rencontrés dans les chapitres précédents. Nous avons proposé une mesure (dérivée du rapport d'approximation différentiel) pour juger de la qualité moyenne des solutions. Nous avons ensuite proposé différents échantillons d'instances possédant des propriétés diverses. Il ressort de cette étude que les deux algorithmes $2-$approchés retournent des solutions de mauvaise qualité. Au contraire, les deux algorithmes gloutons sont très performants et retournent toujours des solutions d'excellente qualité si l'on excepte les graphes spécialement conçus pour les piéger. Nous n'avons pas réussi à trouver d'instance pour laquelle l'algorithme de liste que nous avons proposé dans cette thèse retourne des solutions de mauvaise qualité. Mieux encore, l'espérance de la taille des solutions retournées par ListRight semble être toujours inférieure ou égale à l'espérance de la taille des solutions retournées par Edge Deletion, démontrant, encore une fois, que notre algorithme est tout à fait capable de rivaliser avec les autres algorithmes, et ce malgré un pire cas défavorable.

Chapitre 6

Synthèse et perspectives de la première partie

Ce chapitre fait la synthèse des résultats obtenus dans la première partie de cette thèse et propose un certain nombre de perspectives.

Dans le premier chapitre, nous avons rappelé au moyen de deux algorithmes d'approximation pour le problème du vertex cover que l'évaluation en pire cas ne permet pas de capturer l'intégralité des comportements d'un algorithme sur les différentes classes de graphes. Par exemple, un algorithme peut être contraint de toujours retourner une solution qui atteint son rapport d'approximation en pire cas sur une classe de graphes particulière et de toujours retourner une solution optimale sur une autre. Nous avons aussi souligné que l'influence du non-déterminisme sur le comportement des algorithmes est importante, en montrant que l'algorithme Maximum Degree Greedy obtient de très bons résultats en moyenne sur la classe des graphes spécialement conçus pour le piéger en pire cas, illustrant ainsi le fait que le rapport d'approximation en pire cas provient souvent d'une instance pathologique rare, doublée d'une exécution elle-même pathologique et rare. Tous ces résultats nous montrent qu'il est nécessaire d'utiliser d'autres méthodes d'évaluation que le simple rapport d'approximation en pire cas.

Dans le second chapitre, nous menons une étude comparative entre deux algorithmes de liste, Sorted ListLeft et Sorted ListRight et nous montrons que Sorted ListRight (l'algorithme que nous proposons) retourne toujours une solution de taille inférieure ou égale à celle de Sorted ListLeft, exhibant ainsi une relation de dominance entre deux algorithmes. Ces deux algorithmes sont facilement distribuables, ce qui nous a mené à évaluer leurs performances en nombre de messages échangés et de rounds de communication. Pour ces deux mesures, l'algorithme Sorted ListLeft est plus efficace que l'algorithme Sorted ListRight.

Le troisième chapitre est consacré à l'évaluation de l'influence de la contrainte de tri des listes par degré décroissant sur les performances des algorithmes Sorted ListLeft et Sorted ListRight. Nous montrons que le relâchement de cette contrainte va dégrader le pire cas des deux algorithmes mais qu'elle permet de meilleures performances moyennes. Plus précisément, nous montrons que le rapport d'approximation moyen de ListRight (Sorted ListRight lorsqu'on lui applique des listes non triées) est toujours inférieur à 2, ce qui montre que cet algorithme peut rivaliser avec les meilleurs algorithmes existants. Nous poursuivons cette étude sur les graphes aléatoires de Erdös-Renyi pour déterminer si les mauvaises performances moyennes de ListLeft sur certaines classes

de graphes influencent ses performances au regard de l'ensemble des graphes possibles. Les résultats montrent que la différence de performance moyenne entre les deux algorithmes n'est pas négligeable.

Ayant montré que ListRight est nettement meilleur que ListLeft, nous avons poursuivi notre étude comparative en déterminant l'espérance exacte de la taille des solutions retournées par six algorithmes sur une classe de graphes particulière : les chemins. Une fois leurs espérances connues de manière analytique, nous avons déterminé expérimentalement la loi de distribution des tailles des solutions retournées par chacun de ces algorithmes. Cette étude approfondie du comportement des algorithmes sur une classe particulière nous a permis de les classer en fonction de leurs performances. Dans ce classement, les deux algorithmes 2−approchés se retrouvent aux deux dernières places. Ces mauvaises performances proviennent du fait que le comportement de ces deux algorithmes est particulièrement inadapté aux chemins.

Cette constatation nous a poussés à mener une étude comparative expérimentale de la qualité des solutions retournées par les différents algorithmes rencontrés au cours de cette thèse lorsqu'on les applique sur différentes classes de graphes. Pour cela, nous avons défini une mesure qui capture la qualité moyenne des solutions retournées par un algorithme. Cette mesure est dérivée du rapport d'approximation différentiel et place la moyenne de la taille des solutions (obtenues par un algorithme sur une instance particulière) sur l'intervalle des solutions possibles. Il ressort de cette étude expérimentale que les deux algorithmes 2−approchés retournent des solutions de mauvaise qualité contrairement aux autres algorithmes. L'algorithme Greedy Independent Cover, qui possède un mauvais rapport d'approximation en pire cas, domine nettement les autres algorithmes sur l'ensemble des instances utilisées dans nos échantillons de tests et retourne pratiquement toujours des solutions d'excellente qualité. Cependant, il est possible de construire des instances pour lesquelles il retournera obligatoirement des solutions de mauvaise qualité, ce qui n'est pas le cas de l'algorithme ListRight. Même si ListRight n'est pas le meilleur algorithme, il retourne *toujours* des solutions de bonne qualité et il semble être toujours meilleur en moyenne que l'algorithme Edge Deletion, ce qui fait donc de ListRight un très bon algorithme.

Perspectives

Un certain nombre de questions restent ouvertes. Nous avons vu tout au long de cette partie que l'algorithme ListRight retourne des solutions de bonne qualité, contrairement à l'algorithme Edge Deletion et nous avons posé la conjecture suivante :

Conjecture : Pour tout graphe G, on a : $\mathbb{E}(\text{ListRight}(G)) \leq \mathbb{E}(\text{Edge Deletion}(G))$.

La preuve que cette conjecture est vraie ou fausse n'est pas simple, car le fonctionnement des deux algorithmes est très différent et un raisonnement par récurrence semble difficile. Une piste à explorer est celle, déjà évoquée dans les chapitres précédents, des algorithmes qui retournent la même solution lorsqu'ils considèrent les sommets dans le même ordre. En effet, nous avons montré que les algorithmes ListRight, *LRGlouton* et *online* retournent des solutions identiques lorsqu'ils considèrent les sommets d'une instance dans le même ordre. Cette propriété nous a permis, par exemple, de démontrer que pour tout graphe l'algorithme ListRight est au plus 2−approché en moyenne. La découverte d'algorithmes retournant la même solution que Edge Deletion pourrait permettre de prouver ou d'infirmer la conjecture. Plus généralement, la découverte de groupes d'algorithmes retournant la même solution peut permettre de prouver des propriétés de façon aisée et

constitue donc un intérêt majeur pour la compréhension du comportement des algorithmes.

Une autre perspective concerne les structures responsables du mauvais rapport d'approximation en pire cas de certains algorithmes. Lorsque l'on considère un graphe de pire cas pour un algorithme donné, on peut généralement voir qu'il s'agit d'une instance facile à résoudre, comme par exemple les graphes $Anti - MDG$ qui sont résolus de manière exacte par l'algorithme Greedy Independent Cover. La kernelization est un ensemble de techniques de réduction qui vise à réduire la taille de l'instance à traiter en lui appliquant une phase de pré-traitement (les principales techniques de kernelization sont présentées dans [AKCF$^+$04]). Par exemple pour le problème du vertex cover, Si x est un sommet isolé, alors x n'appartient à aucune solution optimale. Répondre à la question « Existe-t-il un vertex cover de taille k sur le graphe $G = (V, E)$? » revient à répondre à la question « Existe-t-il un vertex cover de taille k sur le graphe $G = (V - \{x\}, E)$? ». De la même façon, si x est un sommet de degré 1 voisin de y, alors il existe une solution optimale contenant y et il ne reste plus qu'a répondre à la question « Existe-t-il un vertex cover de taille $k - 1$ sur le graphe $G = (V - \{x, y\}, E)$? ».

Ces techniques permettent de réduire, en temps polynomial, un graphe G en un graphe G' tel que si on note k la taille de la solution optimale pour G', alors G' ne contient pas plus de $2k$ sommets, plaçant ainsi tous les algorithmes sur un pieds d'égalité en terme de rapport d'approximation en pire cas. Comme il existe une conjecture forte qui dit qu'il n'existe pas d'algorithme a rapport d'approximation en pire cas constant plus petit que 2, sauf si $P = NP$, cela signifie qu'il est improbable qu'on puisse un jour trouver un algorithme qui soit meilleur en ordre de grandeur que celui qui consiste à appliquer la phase de kernelization, puis de sélectionner tous les sommets du graphe réduit. On se doute que sélectionner tous les sommets est une mauvaise solution et il serait intéressant d'utiliser les algorithmes d'approximations après le pré-traitement. Il est donc nécessaire d'effectuer une analyse des performances expérimentales des différents algorithmes sur les graphes résultants. Une perspective intéressante serait de déterminer une méthode permettant de générer tous les graphes résultants de la kernelization.

Une autre perspective intéressante est de poursuivre les travaux menés dans cette thèse sur d'autres problèmes que le vertex cover. Par exemple, le problème du vertex cover est lié au problème de l'ensemble indépendant de taille maximum dans le sens où une solution de l'un mène à une solution de l'autre. De façon étonnante, on connaît des algorithmes d'approximation à facteur constant pour le problème du vertex cover mais pas pour celui de l'ensemble indépendant. L'évaluation des performances des algorithmes en moyenne peut éventuellement permettre d'obtenir des résultats similaires à ceux obtenus pour l'algorithme ListRight, et donc de gagner un ordre de grandeur en proposant un algorithme dont le facteur d'approximation moyen est constant. Plus généralement, l'évaluation en moyenne permet d'obtenir des résultats surprenants, comme nous l'avons vu dans cette partie, et son utilisation devra être généralisée que ce soit pour des problèmes d'optimisation combinatoire ou dans d'autres branches de la recherche en informatique.

Deuxième partie

Évaluation en moyenne du nombre de reconstructions d'un processus de mise-à-jour d'un arbre de connexion

Chapitre 7

Analyse du nombre de perturbations lors du maintien d'un arbre de connexion de faible diamètre

7.1 Introduction

L'étude et la mise en œuvre des systèmes répartis font intervenir de nombreux sous-domaines ou technologies de l'informatique. Citons les *réseaux* pour assurer le transport fiable et rapide de l'information entre les diverses machines composant le système sur des réseaux physiques hétérogènes, les techniques d'estampillage pour assurer la cohérence des informations échangées, la *sécurité* pour garantir la confidentialité des échanges ou le respect des règles d'accès. On trouve aussi des langages et des plate-formes pour les mettre en œuvre de manière la plus simple et la plus modulaire possible.

Tous ces progrès font que de grands parcs de machines peuvent, ou pourront dans un proche avenir, être utilisés pour des *services* de diverses natures. Les services d'informatique répartie (comme le partage des ressources au sens large du terme) et les services de télécommunication se font sur les mêmes machines et les mêmes réseaux physiques. Par exemple, une machine branchée sur le réseau peut servir de « ressource CPU » pour des applications du type « grilles de calculs » lorsque sa charge d'utilisation est faible et servir aussi à son propriétaire pour accéder à divers forums de discutions ou d'échanges d'informations. Le matériel, les OS, les réseaux doivent pouvoir supporter toutes ces utilisations.

Cependant, un système réparti est, à la base, un ensemble de machines « plongées » dans un réseau physique. Ce réseau, mondial, est maintenant immense et chaque machine, n'importe où dans le monde, peut potentiellement « utiliser » un service donné (en tout cas ceux qui sont « ouverts » comme par exemple un jeu en réseau, un forum d'échanges ou une application de partage de ressources informatiques) plus ou moins longtemps. Or, comme tous ces services répartis connectent un grand nombre d'utilisateurs, un contrôle centralisé n'est en général pas envisageable. L'information échangée (aussi bien les données que les messages de contrôle) doit circuler de manière efficace et pérenne entre tous ces individus. Pour cela, des *chemins* de transport « performants » de l'information doivent être établis et exploités. On aboutit ainsi rapidement à des problèmes d'algorithmique de (grands) graphes. La situation est alors compliquée par plusieurs facteurs.

1. Certains de ces problèmes sont intrinsèquement complexes (problèmes NP-complets par exemple).

2. Disposer d'un « bon » algorithme pour le résoudre ne suffit pas toujours. En effet ces algorithmes nécessitent certaines conditions préalables qui ne sont pas toujours vérifiées en pratique :
   (a) Ils ont souvent besoin de connaître l'intégralité du graphe sur lequel il travaillent. Cette connaissance globale n'est pas toujours disponible.
   (b) Ils construisent une solution mais n'ont pas souvent la capacité de s'adapter aux divers changements qui peuvent se présenter.

Ces limitations font que des solutions algorithmiquement simples sont mises en œuvre : des heuristiques de graphes simples à répartir dont on espère qu'elles vont donner de bons résultats. Ce dernier point est central et pourtant leur *performance* est souvent simplement testée sur quelques jeux d'essais plus ou moins représentatifs. Bien que ces tests soient utiles, nous prétendons qu'ils ne sont pas suffisants dans des services impliquant des quantités importantes d'utilisateurs qui vont et viennent dans des réseaux non complètement connus. Des études *analytiques* doivent et peuvent maintenant être menées, notamment grâce aux progrès dans le domaine des algorithmes d'approximation (qui permet de contourner la barrière de la NP-complétude dans de nombreux problèmes de graphes), des algorithmes on-line (qui permettent de gérer les événements qui arrivent au fil de l'eau) et les études en moyennes qui permettent de prouver des éléments de la performance d'un algorithme. Dans ce chapitre nous présentons quelques uns de nos résultats récents ainsi que des perspectives qui utilisent (explicitement ou implicitement) ces domaines.

Le problème spécifique que nous étudions est décrit (modélisé) précisément dans la section 7.1.1. De manière informelle, nous nous intéressons à la façon d'organiser les connections entre des machines réparties dans un vaste réseau, modélisé par un *graphe*. Nous nous concentrons ici sur la construction d'un *arbre* contenant tous les participants. Cet arbre peut alors servir de support pour l'échange de l'information entre les membres et peut être vu comme un « bus » utilisé par « inondation/sélection ». L'absence de cycle fait que les informations ne bouclent pas dans cette structure et qu'ainsi on peut éviter un mécanisme complexe de contrôle de diffusion des messages. Notons que l'on ne cherche pas ici à faire un arbre de multicast pour une seule source mais dans notre étude chaque participant peut être source et récepteur. Ainsi, la distance dans l'arbre entre tous les participants doit être aussi petite que possible. Nous nous focalisons plus précisément ici sur la maîtrise de la distance maximum (nommée *diamètre*) entre les participants.

La maîtrise du diamètre de l'arbre est rendue difficile notamment par le fait que dans ce type de service de connexions, des participants (appelés *membres* dans la suite) nouveaux peuvent venir se connecter et des participants actuels peuvent demander à quitter le service. Ces « entrée/sorties » vont avoir un impact sur la structure de l'arbre qui devra donc être *adapté, mise à jour* après chacun de ces événements. Il conviendra néanmoins de toujours faire ces mises-à-jour de manière simple, tout en assurant le *niveau de qualité* désiré. La contrainte de qualité va faire que, parfois, tout ou partie de l'arbre courant doit être reconstruit. Si ces reconstructions modifient des routes de l'arbre courant, les communications entre les membres seront perturbées pendant la modification. Dans cet chapitre nous proposons d'étudier le nombre de telles perturbations.

Nous supposons ici que nous ne connaissons pas à l'avance les événements d'entrée/sortie de membres qui arrivent donc au fil de l'eau, de manière *on-line*. Nous montrons dans la section 7.2 que dans le pire des cas, le nombre de telles perturbations est linéaire en le nombre d'événements. Ce résultat en pire cas est intrinsèquement lié aux contraintes que nous imposons et est valable pour *n'importe quel algorithme* de mise-à-jour les respectant. Il s'agit donc ici d'une limitation fondamentale. Cependant cette limitation est assez artificielle dans la mesure où le système servant

à la décrire et la suite des événements qui conduisent à cette situation sont très spécifiques et *a priori* rares. Cette rareté apparente nous a conduit à nous poser le problème de l'évaluation de ce paramètre *en moyenne*. Dans la section 7.3 nous proposons un procédé simple de mise-à-jour et nous *démontrons* (en utilisant des techniques de marches aléatoires) qu'il induit en moyenne un nombre au plus proportionnel en le *logarithme* du nombre d'événements ce qui fait de lui un procédé intéressant à déployer.

7.1.1 Modélisation, contraintes et critère d'évaluation

Nous représentons le réseau par un graphe $G = (V, E, w)$ non orienté, connexe et pondéré. V est l'ensemble des *sommets* (représentant les nœuds du réseau), E l'ensemble des *arêtes* (représentant l'ensemble des liens physiques du réseau) et w une fonction qui, à chaque arête $e \in E$, associe un *poids* $w(e)$ strictement positif, (représentant le temps de transmission à travers le lien physique modélisé par l'arête e). Dans notre problème, le graphe G et un *groupe initial* M de membres sont donnés en entrée. Par exemple, si nous considérons une réunion de membres sur un réseau (représenté par G), ce groupe initial M représente l'ensemble des participants présents dès le début de la réunion.

L'ensemble des membres M est un sous-ensemble non vide des sommets de G ($M \subseteq V$ et $M \neq \emptyset$). Nous supposons que les membres à ajouter (resp. retirer) ne sont pas connus à l'avance. Nous nous plaçons donc dans le cadre d'une étude *on-line* (voir [BEY98, FW98] pour des références sur les algorithmes on-line en général). Soulignons que le graphe G est statique (c'est-à-dire que ni les sommets, ni les arêtes de G ne peuvent apparaître ou disparaître) ; seule l'appartenance d'un sommet au groupe courant peut changer. Nous allons maintenant introduire les notations qui vont nous permettre de représenter ces changements. Pour simplifier, nous appellerons *requête* l'événement qui consiste soit à ajouter un nouveau membre, soit à retirer un membre déjà présent dans le groupe courant M. Notons que nous ne nous préoccupons pas ici ni de la manière dont un membre peut exprimer son désir de quitter le groupe (ou la manière dont un nouveau membre peut exprimer sa volonté de participer au groupe) ni de questions de sécurité ou d'autorisations liées à cela. Pour nous ici, une requête vient d'un membre qui a reçu toutes les autorisations éventuelles pour faire ce qu'il veut faire et cette requête doit être traitée.

S'il s'agit de l'ajout du sommet $u \in V \backslash M$, alors on obtient, lorsque cette nouvelle requête d'ajout est traitée, $M \leftarrow M \cup \{u\}$. S'il s'agit du retrait du sommet $u \in M$, alors on obtient, lorsque cette nouvelle requête de retrait est traitée, $M \leftarrow M \backslash \{u\}$. Nous noterons $r_1, \ldots, r_h$ une séquence de h requêtes on-line.

La première contrainte que nous imposons concerne la nature de la structure que nous devons construire à chaque étape pour connecter l'ensemble des sommets appartenant au groupe courant. À chaque étape de construction (donc dès la première étape de couverture du groupe initial, puis ensuite, après chaque nouvelle requête), nous imposons que la structure $T = (V_T, E_T)$ couvrant le groupe courant M (c'est à dire tel que $M \subseteq V_T \subseteq V$) soit un *arbre* élagué, c'est-à-dire que chaque feuille de T est un sommet du groupe courant M (il n'y a donc pas de branches inutiles, dites *branches mortes*, pour la couverture de M dans T). Pour tous sommets u et v appartenant à M, on notera $d_T(u, v)$ la *distance* entre u et v dans T, c'est-à-dire la somme des poids des arêtes d'un chemin de poids minimum entre u et v dans T. Comme nous l'avons indiqué dans l'introduction, nous imposons que notre structure soit un arbre dans le but de simplifier les mécanismes de routage et de duplication des informations à faire circuler entre les membres du groupe.

Pour évaluer la *qualité* d'un arbre, nous nous intéressons ici au diamètre du groupe courant dans

l'arbre. En effet, si le poids d'un lien (une arête du graphe G) représente le temps de transmission d'une information à travers ce lien, alors donner une garantie sur le diamètre permet d'assurer une certaine qualité de communication aux membres utilisateurs de la structure de communication en termes de latence maximum.

Définition 36 (Diamètre d'un groupe) *Le* diamètre *du groupe M dans l'arbre T couvrant M est défini par $D_T(M) = \max\{d_T(u,v) : u,v \in M\}$.*

Si on veut optimiser la latence maximum dans un groupe M, on doit faire en sorte que le diamètre du groupe dans l'arbre couvrant M soit le plus petit possible après la prise en compte de chaque requête. Imposer que le diamètre soit optimal est une condition très forte, difficile à satisfaire en réparti. Nous préférons la relâcher pour garantir que le diamètre de l'arbre courant ne soit pas trop éloigné de l'optimal. Nous exprimons cette exigence de qualité sur le diamètre que nous imposons aux arbres successifs avec la notion de *contrainte de qualité*.

Définition 37 (Contrainte *qualité*) *Soit $c \geq 1$ une constante quelconque, représentant le niveau de qualité désiré pour le diamètre. Soit* A *un algorithme de mise-à-jour d'un arbre de connexion qui construit à chaque étape un arbre T couvrant le groupe courant M. À chaque étape, on considère T^* un arbre optimal pour le diamètre couvrant le groupe courant M, c'est-à-dire un arbre tel que $D_{T^*}(M) = \min\{D_{T'}(M) : T'$ est un arbre couvrant $M\}$. L'algorithme* A *satisfait la contrainte* qualité *avec un* niveau *c si et seulement si, à chaque étape de construction, on a $D_T(M) \leq c \cdot D_{T^*}(M)$.*

L'interprétation de la contrainte *qualité* est la suivante. Un algorithme A respecte la contrainte *qualité* avec un niveau c (où c est une constante) si à chaque étape, il construit un arbre T couvrant M induisant un diamètre au plus c fois plus grand que le diamètre de M dans le meilleur arbre possible. On peut voir la *contrainte qualité* comme une façon de fixer sous forme de *contrainte* le rapport d'approximation que doit respecter *à chaque étape* de construction un algorithme qui répond à notre problème.

Si nous n'imposons que les deux contraintes *arbre* et *qualité*, résoudre le problème que nous nous posons est simple, mais peu intéressant (il suffit en effet de reconstruire totalement l'arbre courant à chaque requête). La difficulté (et l'intérêt de cette étude) vient du fait que nous voulons minimiser les perturbations dans l'arbre courant tout en garantissant qu'à chaque étape de construction, les contraintes *arbre* et *qualité* sont respectées. Cela implique qu'un algorithme sera évalué en fonction des perturbations qu'il va induire dans l'arbre en construction. Afin de quantifier précisément ces perturbations, nous définissons maintenant ce que nous appelons une *étape critique*.

Définition 38 (Étape critique) *Soit r_i la $i^{ème}$ requête on-line (ajout ou retrait d'un membre). Soit $T = (V_T, E_T)$ l'arbre courant construit juste avant le traitement de la requête r_i et soit $T' = (V_{T'}, E_{T'})$ l'arbre construit juste après le traitement de la requête r_i. Si r_i est une requête d'ajout (resp. de retrait), alors l'étape i est une* étape critique *si $E_T \not\subseteq E_{T'}$ (resp. $E_{T'} \not\subseteq E_T$).*

Nous distinguons les étapes critiques des autres car elles génèrent d'importants changements dans l'arbre courant. En effet, un grand nombre de routes entre membres déjà présents dans le groupe courant doivent être changées après une étape critique. Toutes les tables de routage des sommets de connexion peuvent potentiellement être modifiées. Non seulement cette mise-à-jour génère un surplus de trafic important, mais perturbe également les communications entre les membres du

groupe courant déjà en cours. En revanche, un ajout (resp. retrait) simple (c'est-à-dire non critique) nécessite uniquement l'ajout (resp. le retrait) d'un chemin dans l'arbre courant. Il s'agit donc de changements locaux. La mise à jour des tables de routage ne nécessite que la diffusion de l'identité du nouveau membre (resp. l'information de disparition du membre retiré) dans l'arbre. Il n'y a donc pas de re-routage nécessaire entre les membres du groupe courant. Le nombre d'étapes critiques doit donc être minimisé.

7.1.2 État de l'art

La plupart des références bibliographiques que nous allons donner ici ne font référence qu'à des algorithmes de graphe (non répartis) et traitent de problématiques proches des nôtres (on trouvera dans [BBL06] un algorithme réparti pour construire un arbre couvrant un groupe donné mais qui ne traite pas de la question du traitement des requêtes d'ajouts/retraits).

Le problème que nous étudions a été exploré essentiellement du point de vue du *poids* de l'arbre courant (c'est-à-dire la somme des poids des arêtes de l'arbre). La version off-line de ce problème (c'est-à-dire lorsque les données sont intégralement connues dès le départ) est connue sous le nom du problème de l'arbre de Steiner. Étant donné qu'il s'agit d'un problème NP-complet (voir [GJ79]), des algorithmes d'approximations ont été proposés pour résoudre ce problème (par exemple dans [APMS$^+$99, Hoc97]).

B. Waxman a été le premier à présenter la version on-line de l'arbre de Steiner [Wax88]. Dans cet article, il a divisé le problème en deux catégories : un premier modèle dans lequel les changements de routes dans l'arbre ne sont pas autorisés (c'est-à-dire qu'aucune étape n'est critique), et un deuxième modèle dans lequel les changements dans l'arbre courant sont autorisés (correspondant au modèle que nous étudions ici). Puis, dans [IW91], M. Imaze et B. Waxman ont proposé un algorithme pour chaque modèle. Pour le modèle sans changements dans l'arbre courant, les auteurs se sont limités à l'étude des séquences d'ajouts on-line (de nouveaux membres peuvent intégrer le groupe courant au fur et à mesure, mais ne peuvent pas le quitter). Ils prouvent que l'algorithme qu'ils proposent a un rapport de compétitivité (rapport d'approximation on-line) en $O(\log h)$ pour le critère poids (où h est le nombre de sommets ajoutés). Ils montrent également qu'il n'existe pas d'algorithme sans reconstruction dont le rapport de compétitivité est meilleur qu'une fonction en $\Omega(\log h)$. Pour le modèle où les changements sont autorisés, les auteurs proposent un algorithme traitant le cas des ajouts et des retraits on-line mêlés. Cette stratégie a un rapport de compétitivité constant pour le critère poids et induit un nombre total de changements de routes en $O(h\sqrt{h})$. En revanche, les auteurs ne donnent pas de garantie sur le nombre d'*étapes critiques* (toutes les étapes sont donc potentiellement critiques). Nous pouvons maintenant diviser les travaux existants en deux parties (comme dans [Wax88]).

D'une part, dans [AA92, AAB96, AAA$^+$06, Ang07] (par exemple), le modèle où les changements dans l'arbre ne sont pas autorisés est considéré. Dans [AA92], les auteurs donnent une borne inférieure en $\Omega(\frac{\log h}{\log \log h})$ sur le rapport de compétitivité de tout algorithme résolvant le problème de l'arbre de Steiner on-line (donc pour le critère poids) dans un plan Euclidien. Dans [AAB96], les auteurs considèrent le problème de l'arbre de Steiner généralisé : étant donné un ensemble de paires de sommets, il s'agit de trouver un sous-graphe de poids minimum tel que chaque paire de sommet est connectée par le sous-graphe construit. Un algorithme dont le rapport de compétitivité est en $O(\log h)$ est proposé pour la version traitant les ajouts on-line de ce problème. Dans [AAA$^+$06], des bornes supérieures et inférieures sont obtenues pour différentes variantes du problème de l'arbre de Steiner on-line généralisé. Enfin, dans [Ang07], des bornes supérieures et inférieures linéaires en

nombre de paires ajoutées sont obtenues pour le problème de l'arbre de Steiner on-line généralisé sur les graphes orientés.

D'autre part, dans [GM02, SMM99, ZM06] (par exemple), le modèle où les changements dans l'arbre sont autorisés est envisagé. Dans [GM02], le but est de minimiser simultanément le poids de l'arbre courant et la distance à partir d'un sommet racine vers tous les autres sommets du groupe courant. Seule la version on-line avec ajouts est traitée. Les auteurs proposent alors une stratégie induisant au plus un changement élémentaire par étape et dont le rapport de compétitivité est simultanément en $O(\log h)$ pour le critère poids et constant pour le critère distance à la racine. Dans [ZM06], les auteurs proposent plusieurs algorithmes pour la construction de structures de multicast (d'une source vers plusieurs destinations) dans le cas d'ajouts et de retraits on-line de destinations. Leur but est de trouver un équilibre entre le poids de la structure construite et le nombre de chemins réarrangés. Leurs algorithmes sont ensuite évalués par des simulations. Dans [AIS03, LC05, SMM99], des méthodes traitant simultanément les ajouts et les retraits on-line pour l'optimisation de plusieurs critères (dont le poids et le diamètre) sont proposées. Ces méthodes sont ensuite évaluées par des simulations. Aucune borne supérieure analytique n'est proposée, ni pour les différents rapports de compétitivité, ni pour le nombre de changements induits. Nous soulignons que dans [GM02, IW91, SMM99], seuls le nombre de changements total de routes est pris en compte pour mesurer le niveau de perturbation due aux changements dans l'arbre courant.

À notre connaissance, il n'existe pas de travaux antérieurs aux nôtres qui considèrent le nombre d'*étapes critiques* comme critère d'évaluation d'un algorithme.

7.2 Nombre d'étapes critiques au pire cas

Dans cette section, nous prouvons que pour tout algorithme respectant les contraintes *arbre* et *qualité*, pour tout h suffisamment grand, il existe un graphe et une séquence de requêtes qui induit un nombre d'étapes critiques linéaire en h (où h est le nombre de requêtes). Pour montrer ce résultat, nous allons utiliser un *adversaire adaptatif*. Il s'agit d'une notion classique en algorithmique on-line (voir [BEY98]). L'idée est la suivante : pour montrer un résultat d'impossibilité, un *adversaire* va révéler au fur et à mesure les données du problème de manière à piéger n'importe quel algorithme on-line. L'adversaire va s'*adapter* en révélant chaque nouvelle donnée à traiter en fonction du comportement de l'algorithme aux étapes précédentes. Ainsi, l'usage d'un tel adversaire permet d'obtenir un résultat universel d'impossibilité sur la performance de tout algorithme on-line dans le *pire cas*.

Théorème 39 *Soit $c \geq 1$ une constante quelconque (représentant le niveau de qualité pour le diamètre). Pour tout algorithme* A*, pour tout h suffisamment grand, il existe un graphe, un groupe initial et une séquence de h requêtes on-line tels que* A *induit un nombre linéaire d'étapes critiques (un nombre d'étapes critiques en $\Omega(h)$).*

Le théorème 39 montre que le maintien d'une qualité constante pour le diamètre induit un nombre d'étapes critiques linéaire en nombre de requêtes on-line. Ce résultat est négatif, puisqu'il montre que tout algorithme on-line (réparti ou centralisé) doit reconstruire un nombre de fois comparable en ordre de grandeur à celui de la solution triviale consistant à casser l'arbre à chaque étape pour le reconstruire totalement.

Pour ne pas s'arrêter à ce résultat négatif, plusieurs approches peuvent être envisagées. Une

première approche consiste à ne pas considérer n'importe quelle séquence de requêtes, mais seulement les séquences de requêtes constituées soit uniquement d'ajouts de nouveaux membres, soit uniquement de retraits de membres déjà présents dans le groupe courant. Cela signifie que sont traitées seulement les applications où deux phases bien distinctes sont identifiables (par exemple une phase où le groupe se forme de manière incrémentale, composée uniquement de requêtes d'ajouts, puis une phase où le groupe décroît, composée uniquement de requêtes de retraits). Bien sûr, cette approche n'est pas totalement convaincante, puisqu'elle restreint la validité des résultats obtenus à ces cas particuliers. Cette étude a néanmoins été menée dans [TL06b]. Lorsque la séquence de requêtes on-line n'est constituée que d'ajouts, on obtient très facilement un algorithme qui garantit une constante de qualité $c = 2$ pour le diamètre sans aucune étape critique. Il suffit pour cela d'incrémenter à chaque nouvelle requête d'ajout l'arbre courant en lui ajoutant un plus court chemin entre une racine fixe (un membre quelconque du groupe de départ) et le nouveau membre. Lorsque la séquence de requêtes on-line n'est constituée que de retraits, la situation est plus délicate à gérer. Dans [TL06b], un algorithme (beaucoup plus sophistiqué que dans le cas de requêtes d'ajouts uniquement) garantissant une constante de qualité $c = 4$ pour le diamètre et induisant $O(\log h)$ étapes critiques (avec h le nombre de requêtes de retraits) est proposé. Il est également prouvé dans [TL06b] que tout algorithme garantissant une qualité c constante pour le diamètre induit $\Omega(\log h)$ étapes critiques (l'algorithme proposé est donc optimal en ordre de grandeur pour le nombre d'étapes critiques dans le pire cas).

La deuxième approche consiste à évaluer le nombre d'étapes critiques d'un algorithme pour une séquence quelconque de requêtes (ajouts et retraits mêlés) non pas dans le pire des cas, mais en moyenne. Cette approche se justifie doublement. D'une part parce que le résultat en pire cas est négatif (cf théorème 39), d'autre part car la situation qui permet d'obtenir ce résultat très négatif (l'adversaire adaptatif utilisé dans la preuve du théorème 39) correspond à un cas pathologique peu représentatif des situations réseaux qui ont lieu en pratique.

7.3 Évaluation en moyenne du nombre de reconstructions d'un algorithme de mise-à-jour

Dans cette partie, nous proposons en section 7.3.1 un algorithme qui respecte les contraintes arbre et qualité. En section 7.3.2 nous présentons le modèle probabiliste utilisé pour l'évaluation en moyenne du nombre d'étapes critiques de l'algorithme en sections 7.3.3 et 7.3.4.

Une version préliminaire de ces travaux a été publiée dans [DLT07]. Ce chapitre est en cours de soumission pour le journal français Technique et Science Informatiques.

7.3.1 Description du processus de mise-à-jour étudié.

Pour connecter le groupe initial, un membre r est choisi arbitrairement et un arbre des plus courts chemins enraciné en r couvrant tous les membres du groupe est construit. Soit T l'arbre courant de racine r. Considérons la prochaine opération, qui peut être soit un ajout, soit un retrait. S'il s'agit de l'ajout d'un nouveau membre u, l'arbre est mis à jour en étendant l'une de ses branches pour inclure le nouveau membre u (si u est déjà un noeud de T, il est juste inclus dans le groupe et aucune modification structurelle de T n'est effectuée). Cette extension doit être effectuée de telle sorte que T est un arbre des plus courts chemins, enraciné en r, couvrants tous les membres du

groupe (cette extension est toujours possible). Considérons maintenant le retrait d'un membre u du groupe courant. Si u est une feuille de T, il suffit d'élaguer l'arbre (c'est-à-dire libérer le chemin qui connecte u au reste de l'arbre). Si u n'est pas une feuille, la structure de T n'est pas modifiée et u est simplement retiré du groupe, devenant un nœud de connexion pour le reste du groupe. Si le sommet quittant le groupe est r (c.a.d. $u = r$), il n'est plus possible de garantir la contrainte de qualité pour le diamètre. L'arbre courant est alors défait et un (nouvel) arbre de plus courts chemins est construit en choisissant (arbitrairement) une nouvelle racine parmi les membres du groupe restant. Une telle étape est appelée *étape de reconstruction* et constitue une étape critique (cf définition 38). On pourrait envisager de raffiner et de ne reconstruire *que* ce qui est nécessaire dans l'arbre. Cependant, cela ne changera pas l'étude qui suit, ce raffinement peut être difficile à effectuer de manière répartie et cela constituerait malgré tout une étape critique.

Lors de chaque opération, T est bien un arbre couvrant le groupe courant et son diamètre est au plus deux fois celui du groupe courant dans le réseau sous-jacent, ce qui assure une contrainte de qualité de $c = 2$. On note qu'avec ce processus de mise-à-jour une reconstruction n'a lieu *que* lorsque la racine de l'arbre courant se retire du groupe. Par définition de l'algorithme, une étape sera critique (au sens de la définition 38) uniquement si la racine est retirée (et donc que l'on effectue une reconstruction). En effet, lors du retrait d'un sommet autre que la racine (resp. l'ajout d'un sommet quelconque), le nouvel arbre est inclus dans le précédent (resp. inclut le précédent).

Cet algorithme est très simple. Il ne fait appel qu'à des connexions par des plus courts chemins. Il est important de noter que le critère sur lequel s'appuie la prise de décision pour lancer une reconstruction est purement local et est facile à mettre en œuvre : c'est lorsque le membre du groupe qui fait office de racine se retire du groupe que la reconstruction est faite.

Remarque : le fait qu'une opération de retrait concerne la racine est *a priori* indépendant de la topologie du graphe sous-jacent, car la racine est choisie arbitrairement parmi les membres du groupe lors de chaque reconstruction. Ainsi, le graphe sous-jacent n'interviendra pas dans l'évaluation du nombre d'étapes de reconstruction de notre processus.

7.3.2 Modélisation

Nous supposons ici que la probabilité qu'une opération concerne l'ajout d'un nouveau membre ou le retrait d'un membre du groupe est fixe pour toute la durée de l'existence du groupe de communication. On notera p la probabilité d'une opération d'ajout, et $q = (1 - p)$ la probabilité d'une opération de retrait.

Un groupe de communication possède, lors de sa création, au moins un membre et plusieurs machines peuvent créer, toutes ensembles, un groupe de communication. On peut, par exemple, considérer le cas d'un service de réunion virtuelle initiée par plusieurs personnes qui s'étaient données rendez-vous à une heure précise, et où de nouveaux participants viennent se greffer au fur et à mesure, tandis que certains participants déjà présents quittent cette réunion une fois leur intervention effectuée. Les membres d'un groupe font partie d'un graphe sous-jacent que l'on suppose infini, ce qui implique que la taille maximum d'un groupe de communication n'est pas limitée *a priori* (ceci est très utile pour l'analyse et sera discuté dans les commentaires des résultats). Lorsque la taille d'un groupe atteint 0, le groupe meurt (les éventuelles opérations d'ajouts et de retraits reçues ensuite sont toutes ignorées).

Lorsqu'une opération de retrait est effectuée, elle peut *a priori* concerner n'importe quel membre du groupe. Ainsi la probabilité que cette opération concerne un membre en particulier est $\frac{1}{\textit{taille du groupe courant}}$.

Dans l'algorithme que nous proposons, une reconstruction n'a lieu que lorsque le membre identifié comme la racine quitte le groupe de communication. Ainsi, évaluer le nombre d'étapes de reconstructions revient à évaluer le nombre de fois qu'une opération va concerner le retrait de la racine de l'arbre couvrant le groupe courant.

Ainsi, la probabilité de devoir reconstruire l'arbre courant lors de l'exécution d'une opération quelconque est égale à la probabilité q qu'il s'agisse d'une opération de retrait multipliée par la probabilité que cette opération ait pour cible la racine de l'arbre courant, soit $q \cdot \frac{1}{\textit{taille du groupe courant}}$.

Afin de ne pas augmenter artificiellement le nombre de reconstructions effectuées par notre algorithme, nous considérons que lorsqu'un groupe de taille 1 effectue un retrait (et qui concerne donc obligatoirement la racine), il n'est pas nécessaire d'effectuer une reconstruction car la taille du groupe résultant est 0 et le groupe de communication s'éteint.

Dans la suite de ce chapitre, $\mathbb{E}(z,h)$ correspondra à l'espérance du nombre de reconstructions après h opérations d'ajout et/ou de retrait en partant d'un groupe initial de taille z.

7.3.3 Resultat principal : idée de preuve et interprétation

Il est facile de voir que dans le pire cas (c'est-à-dire lorsque chaque opération de retrait concerne la racine courante) notre algorithme effectue un nombre de reconstructions linéaire en le nombre de requêtes. Dans cette partie, nous effectuons une analyse plus fine et nous montrons qu'en moyenne, le nombre de reconstructions effectuées ne dépend pas du nombre de requêtes ou bien en dépend logarithmiquement. Plus précisément, nous avons le théorème suivant :

Théorème 40 *Soit p la probabilité qu'une requête concerne l'ajout d'un nouveau membre, $q = (1-p)$ la probabilité qu'elle concerne un retrait, et z le paramètre initial représentant la taille du groupe de départ. On a :*

- *Lorsque $p < q$, $\mathbb{E}(z,h) \leq \frac{z}{2(1-2p)}$.*
- *Lorsque $p = q$, $\mathbb{E}(z,h) \leq 2^{z-1} \sum_{n=1}^{h+z-1} \frac{1}{n}$.*
- *Lorsque $p > q$, $\mathbb{E}(z,h) \leq \frac{1}{e^{\left(\frac{1}{8}(2p-1)\right)^2}-1} + q\frac{2}{2p-1} \sum_{n=1}^{h} \frac{1}{n}$.*

La preuve détaillée de ce résultat est présentée en section 7.3.4. Nous proposons dans ce qui suit la description de ses grandes lignes, puis une interprétation précisant l'impact de ce théorème dans le cadre des groupes dans les réseaux.

Principes de la preuve

On remarque que la taille du groupe courant ne peut évoluer que de deux manières. En effet, il ne peut que diminuer ou augmenter sa taille de 1, suivant qu'il effectue une opération d'ajout ou de retrait. Dans notre modèle, la probabilité d'effectuer un ajout ou un retrait est indépendante des opérations effectuées précédemment ou des tailles des groupes précédents, ce qui indique un processus markovien. Nous avons utilisé les marches aléatoires et les différents résultats associés (voir [Fel71]) pour déterminer l'évolution du groupe courant.

Ces résultats nous indiquent que si $p < q$, le groupe va atteindre une taille de 0 de façon certaine. En considérant les différents temps où le groupe peut atteindre une taille de 0, et en remarquant que la probabilité d'effectuer une reconstruction est au plus de $q\frac{1}{2}$, on obtient (en utilisant les séries hypergéometriques) que $\mathbb{E}(z,h) \leq \frac{z}{2(1-2p)}$.

Lorsque $p > q$, le groupe va avoir tendance à grandir et n'atteindra presque certainement jamais une taille de 0. En considérant une marche aléatoire qui évolue de la même façon que la taille du groupe, mais qui ne s'arrête pas lorsqu'elle passe par 0, on obtient une majoration de $\mathbb{E}(z,h)$. Cette majoration est composée de deux séries liées au nombre d'opérations, l'une est majorée par une constante (*i.e.* indépendante du nombre h de requêtes) car elle converge, l'autre est logarithmique en h. Plus précisément, on obtient que $\mathbb{E}(z,h) \leq \frac{1}{e^{\left(\frac{1}{8}(2p-1)\right)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$.

Dans le cas où $p = q = \frac{1}{2}$, les différents théorèmes issus des marches aléatoires ne permettent pas de déterminer le comportement de l'évolution de $\mathbb{E}(z,h)$. Nous avons déterminé une expression exacte de $\mathbb{E}(1,h)$ lorsque $p = q = \frac{1}{2}$ et nous avons démontré, grâce à cette expression que $\mathbb{E}(1,h) \leq \sum_{n=1}^{h}\frac{1}{n}$. À partir de cette majoration de $\mathbb{E}(1,h)$ et en généralisant pour toute taille de groupe de départ, on obtient que $\mathbb{E}(z,h) \leq 2^{z-1}\sum_{n=1}^{h+z-1}\frac{1}{n}$.

Interprétation du théorème 40

Le théorème 39 nous indique que tout algorithme devra effectuer, en pire cas, un nombre de reconstructions linéaire en h, le nombre d'opérations d'ajout et/ou de retrait. Pour notre algorithme, il suffit de considérer le cas où chaque opération de retrait concerne la racine courante. Cependant, le théorème 40 montre qu'en moyenne notre algorithme va effectuer un nombre de reconstructions qui est indépendant de h lorsque $p < q$, et qui dépend logarithmiquement de h sinon (car on sait que $\sum_{n=1}^{h}\frac{1}{n} \approx \ln h$ très rapidement). Le théorème 40 montre aussi que $\mathbb{E}(z,h)$ dépend de z, la taille du groupe initial. Cependant, certains services sont toujours initiés par un petit nombre d'individus (forum interactif autour d'un cours universitaire, initié par l'enseignant ou une association étudiante, une conférence téléphonique, initiée par une petite équipe, ...). Dans ces applications, la taille du groupe initial est petite (de l'ordre d'une dizaine de personnes). En considérant ces applications, on peut déduire du théorème 40 que $\mathbb{E}(z,h) \in O(1)$ lorsque $p < q$ et $\mathbb{E}(z,h) \in O(log(h))$ sinon.

Nos résultats, obtenus dans des graphes infinis, restent valables lorsque la taille du graphe sous-jacent possède une taille finie n « importante » et que le nombre d'opérations reste inférieur à $n - z$ (*i.e.* tant qu'aucune évolution possible de la taille du groupe de communication n'est en mesure d'atteindre la taille du graphe sous-jacent les preuves restent valables). Dans une application sur Internet, la taille du réseau sous-jacent est effectivement très grande ; le nombre de machines connectées dans internet est estimé à plusieurs centaines de millions. Nos résultats restent donc théoriquement valables pour des centaines de millions d'opérations (par exemple dans un jeu interactif international initiée par un petit groupe initial).

7.3.4 Preuve du théorème 40

Le théorème 40 se déduit directement des lemmes 44, 47 et 53. Le lemme 44 montre que lorsque $p < q$, $\mathbb{E}(z,h) \leq \frac{z}{2(1-2p)}$, le lemme 47 montre que lorsque $p > q$, $\mathbb{E}(z,h) \leq \frac{1}{e^{\left(\frac{1}{8}(2p-1)\right)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$ et le lemme 53 que quand $p = q$, $\mathbb{E}(z,h) \leq 2^{z-1}\sum_{n=1}^{h+z-1}\frac{1}{n}$.
Les preuves des lemmes 44 et 47 utilisent les marches aléatoires sur $\mathbb{Z}$.
Une marche aléatoire sur $\mathbb{Z}$ est un processus stochastique de type chaine de Markov. Un tel objet est une suite $(X_n)_{n\in\mathbb{N}}$ d'entiers, décrit par un entier relatif X_0 (le paramètre de départ) et une suite de variables aléatoires $(Y_n)_{n\geq 1}$ indépendantes et de même loi de probabilité, toutes à valeur dans $\{-1,1\}$. On définit pour $n \geq 1$, $X_n = X_0 + \sum_{k=1}^{n} Y_k$. Dans un tel processus, la valeur au

temps $n+1$ est celle au temps n plus le $n+1$-ième pas (la variable Y_{n+1}), qui vaut plus ou moins un. Dire que les variables aléatoires $(Y_n)_{n\geq 1}$ sont indépendantes, c'est exprimer le fait qu'au temps n, on effectue un pas -1 (ou de façon plus imagée un pas à gauche si on se représente $\mathbb{Z}$ comme un axe horizontal) ou $+1$ (un pas à droite) indépendamment de ce qui s'est passé auparavant. Dans la suite, on notera p la probabilité d'effectuer un pas à gauche et $q = (1-p)$ la probabilité d'effectuer un pas à droite.

Définition 41 (Marche Ajout-Retrait) *La marche Ajout-Retrait est une marche aléatoire sur $\mathbb{Z}$ et de paramètre de départ $z > 0$. Lorsque la marche Ajout-Retrait atteint 0, elle s'arrête. Cette marche aléatoire correspond à l'évolution de la taille du groupe à connecter en partant d'un groupe initial de taille z.*

Il existe de nombreux résultats sur les marches aléatoires. Les deux théorèmes suivants (que l'on trouve dans [Fel71]) nous seront utiles pour les démonstrations des lemmes 44 et 47.

Théorème 42 *Étant donnée une marche aléatoire sur $\mathbb{Z}$ de paramètre de départ $z > 0$ et en notant q_z la probabilité de passer par 0, on a :*

$$q_z = \begin{cases} 1 & si\ p \leq q \\ \left(\frac{q}{p}\right)^z & si\ p > q \end{cases}$$

Théorème 43 *Considérons une marche aléatoire sur $\mathbb{Z}$ de paramètre de départ $z > 0$. Soit T le moment du premier passage en 0 de cette marche aléatoire. $\forall h \geq 0$, on a :*

$$P(T = z+2h) = \frac{z}{z+2h}\binom{z+2h}{z+h}p^h q^{z+h}$$

Lorsque $p < q$, $\mathbb{E}(z,h) \leq \frac{z}{2(1-2p)}$

Lemme 44 *Soit $\mathbb{E}(z,h)$ l'espérance du nombre de reconstructions en partant d'un groupe de taille z après h requêtes. Lorsque $p < q$, $\mathbb{E}(z,h) \leq \frac{z}{2(1-2p)}$.*

La preuve du lemme 44 nécessite le lemme suivant :

Lemme 45 $\sum_{h=0}^{\infty}\binom{z+2h}{z+h}X^h = \frac{2^z}{\sqrt{1-4X}\left(1+\sqrt{1-4X}\right)^z}$

Preuve. La preuve de ce lemme utilise les séries hypergéométriques, qui constituent un outil puissant, car il existe un grand nombre de formules permettant de les manipuler (voir [AS64]).

Définition 46 *La forme standard d'une série hypergéométrique généralisée est :*
${}_rF_s(a_1,\ldots,a_r;b_1,\ldots,b_s;z) = \sum_{k=0}^{\infty}\frac{(a_1)_k(a_2)_k\cdots(a_r)_k}{(b_1)_k(b_2)_k\cdots(b_s)_k}\frac{z^k}{k!}$*, où z est le point d'évaluation de la série, r et s sont respectivement le nombre de paramètres hauts et bas de la série et $(x)_k = x(x+1)\cdots(x+k-1)$ la fonction factorielle croissante (ou symbole de Pochhammer).*

Pour plus de détails sur les séries hypergéométriques, le lecteur est invité a consulter [Gau99]. Pour prouver le lemme, nous avons besoin des deux formules suivantes qui sont disponibles dans [AS64] sous les numéros 15.1.1 et 15.1.14 :

Formule 15.1.1 : ${}_2F_1\left(a,b;c;z\right) =_2 F_1\left(b,a;c;z\right)$.
Formule 15.1.14 : ${}_2F_1(a,\frac{1}{2}+a;2a;z) = 2^{2a-1}(1-z)^{-\frac{1}{2}}\left(1+(1-z)^{\frac{1}{2}}\right)^{1-2a}$.

Par définition des séries hypergéométriques, nous avons :

$$ {}_2F_1\left(1+\frac{1}{2}z,\frac{1}{2}+\frac{1}{2}z;1+z;4X\right) = \sum_{h=0}^{\infty}\frac{(1+\frac{1}{2}z)_h\cdot(\frac{1}{2}+\frac{1}{2}z)_h}{(1+z)_h}\cdot\frac{(4X)^h}{h!} $$

On remarque que

$$ \begin{aligned}(1+\frac{1}{2}z)_h\cdot(\frac{1}{2}+\frac{1}{2}z)_h &= \underbrace{\left(1+\frac{z}{2}\right)}_{\frac{1}{2}(2+z)}\cdot\underbrace{\left(2+\frac{z}{2}\right)}_{\frac{1}{2}(4+z)}\cdots\underbrace{\left(h+\frac{z}{2}\right)}_{\frac{1}{2}(2h+z)}\cdot\underbrace{\left(\frac{1}{2}+\frac{z}{2}\right)}_{\frac{1}{2}(1+z)}\cdot\underbrace{\left(1+\frac{1}{2}+\frac{z}{2}\right)}_{\frac{1}{2}(3+z)}\cdots\underbrace{\left(h-\frac{1}{2}+\frac{z}{2}\right)}_{\frac{1}{2}(2h-1+z)}\\ &= \frac{(z+2h)!}{z!}\frac{1}{2^{2h}}\end{aligned} $$

donc

$$ \sum_{h=0}^{\infty}\frac{(1+\frac{1}{2}z)_h\cdot(\frac{1}{2}+\frac{1}{2}z)_h}{(1+z)_h}\cdot\frac{(4X)^h}{h!} = \sum_{h=0}^{\infty}\frac{(z+2h)!}{z!\,(1+z)_h}4^{-h}4^h\frac{X^h}{h!} = \sum_{h=0}^{\infty}\frac{(z+2h)!}{(z+h)!}\frac{X^h}{h!} = \sum_{h=0}^{\infty}\binom{z+2h}{z+h}X^h $$

Il suffit maintenant de montrer que ${}_2F_1\left(1+\frac{1}{2}z,\frac{1}{2}+\frac{1}{2}z;1+z;4X\right) = \frac{2^z}{\sqrt{1-4X}\left(1+\sqrt{1-4X}\right)^z}$:
D'après la formule **15.1.14**,

$$ {}_2F_1(a,\frac{1}{2}+a;2a;4X) = 2^{2a-1}(1-4X)^{-\frac{1}{2}}\left(1+(1-4X)^{\frac{1}{2}}\right)^{1-2a} $$

En posant $a=\frac{1}{2}+\frac{1}{2}z$ on obtient que

$$ {}_2F_1\left(\frac{1}{2}+\frac{1}{2}z,1+\frac{1}{2}z;1+z;4X\right) = \frac{2^z}{\sqrt{1-4X}\left(1+\sqrt{1-4X}\right)^z} $$

et par la formule **15.1.1** nous avons

$$ {}_2F_1\left(1+\frac{1}{2}z,\frac{1}{2}+\frac{1}{2}z;1+z;4X\right) = \frac{2^z}{\sqrt{1-4X}\left(1+\sqrt{1-4X}\right)^z} $$

Nous avons donc

$$ \sum_{h=0}^{\infty}\binom{z+2h}{z+h}X^h = {}_2F_1\left(1+\frac{1}{2}z,\frac{1}{2}+\frac{1}{2}z;1+z;4X\right) = \frac{2^z}{\sqrt{1-4X}\left(1+\sqrt{1-4X}\right)^z} $$

■

Preuve du lemme 44. Considérons la marche *Ajout-Retrait* (définition 41), de paramètre de départ z, correspondant à la taille du groupe initial. On associe à cette marche un paramètre R qui correspond au nombre de reconstructions effectuées depuis le début de la marche. Ainsi, nous commençons avec $R=0$, puis pour chaque pas à gauche à un instant n, on augmente R de 1 avec probabilité $\frac{1}{X_n}$, avec X_n la valeur de la marche *Ajout-Retrait* après n pas. Notre objectif

est de connaître le comportement asymptotique de R. D'après le théorème 42, la marche 41 finit forcement par passer par 0 (car $p < q$) et ainsi par s'arreter. Soit Ω l'ensemble des moments T où la marche peut passer pour la première fois par 0. On a : $\Omega = \bigcup_{h\in\mathbb{N}} \{T = z + 2h\}$ donc $\mathbb{E}(R) = \sum_{h=0}^{\infty} \mathbb{E}(R|T = z + 2h)P(T = z + 2h)$ or, si $T = z+2h$, il y a eu exactement $z+h$ pas vers la gauche. A chaque fois, $P(R \textit{ augmente de } 1 \textit{ au pas } n) = \frac{1}{X_n} \leq \frac{1}{2}$ car on effectue une reconstruction uniquement si $X_n > 1$. Donc $\mathbb{E}(R|T = z + 2h) \leq \frac{z+h}{2}$. Nous en déduisons, grâce au théorème 43 que :

$$\mathbb{E}(R) \leq \sum_{h=0}^{\infty} \frac{z+h}{2} \frac{z}{z+2h} \binom{z+2h}{z+h} p^h q^{z+h}$$

$\Rightarrow \mathbb{E}(R) \leq \frac{z}{2} q^z \sum_{h=0}^{\infty} \binom{z+2h}{z+h} (pq)^h$. Par le lemme 45, et en posant $X = pq = p(1-p) = p - p^2$, on obtient que

$$\mathbb{E}(R) \leq \frac{z}{2} q^z \frac{2^z}{\sqrt{1-4pq}\left(1+\sqrt{1-4pq}\right)^z} = \frac{z}{2} q^z \frac{2^z}{(1-2p)\left(1+(1-2p)\right)^z}$$

$$\mathbb{E}(R) \leq \frac{z}{2} \frac{(2q)^z}{(1-2p)\left(2(1-p)\right)^z} = \frac{z}{2} \frac{(2q)^z}{(1-2p)\left(2q\right)^z} = \frac{z}{2(1-2p)}$$

Comme $\mathbb{E}(z,h) \leq \mathbb{E}(z,\infty) = \mathbb{E}(R)$, le résultat est montré. ■

Lorsque $p > q$**,** $\mathbb{E}(z,h) \leq \frac{1}{e^{\frac{1}{8}(2p-1)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$

Lemme 47 $\mathbb{E}(z,h) \leq \frac{1}{e^{\frac{1}{8}(2p-1)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$ *lorsque* $p > q$.

La démonstration de ce lemme utilise le théorème suivant (voir [McD98]) :

Théorème 48 *Soit* $0 < p < 1$*, soient* $W_1, W_2, \ldots, W_n$ *des variables aléatoires indépendantes binaires, avec* $P(W_k = 1) = p$ *et* $P(W_k = 0) = 1 - p\ \forall k$*, et* $S_n = \sum W_k$*. Alors,* $\forall t \geq 0$*,* $P(|S_n - np| \geq nt) \leq 2e^{-2nt^2}$.

Preuve du lemme 47.

Considérons une marche aléatoire sur $\mathbb{Z}$, de paramètre de départ $z \geq 1$, correspondant à la taille du groupe initial. Soit X_n la valeur de cette marche à l'instant n et S_n le nombre de pas à droite parmi les n premiers. On associe à cette marche un paramètre R'_h qui représente le nombre de reconstructions effectuées depuis le début de la marche et on note R_h le nombre de reconstructions effectuées par notre algorithme après h requêtes. Ainsi, nous commençons avec $R'_h = 0$, puis pour chaque pas $i \leq h$ à gauche, on augmente R'_h de 1 avec probabilité $\frac{1}{X_n}$ uniquement si $X_n \geq 1$.

Nous avons $R_h \leq R'_h$ et $\mathbb{E}(R_h) \leq \mathbb{E}(R'_h)$, car la marche à laquelle est associée R'_h ne s'éteint pas si elle passe par 0, cette marche effectue donc plus de pas à gauche que la marche $Ajout - Retrait$. Montrons dans un premier temps que $P(X_n \leq n\frac{2p-1}{2} + z) \leq e^{-n\frac{1}{8}(2p-1)^2}$. On a :

$$X_n = z + \underbrace{S_n}_{\textit{pas a droite}} - (\underbrace{n - S_n}_{\textit{pas a gauche}}) = z + 2S_n - n$$

Alors,

$$\mathbb{E}(X_n) = \mathbb{E}(z + 2S_n - n) = z + 2\mathbb{E}(S_n) - n = z + 2pn - n = z + (2p-1)n$$

et $X_n - \mathbb{E}(X_n) = 2(S_n - pn)$. Donc

$$P(|X_n - \mathbb{E}(X_n)| \geq nt) = P(|S_n - pn| \geq \frac{nt}{2})$$

et par le théorème 48, on a : $P(|X_n - \mathbb{E}(X_n)| \geq nt) \leq 2e^{-\frac{nt^2}{2}}$, c'est-à-dire $P(|X_n - (z + (2p-1)n)| \geq nt) \leq 2e^{-\frac{nt^2}{2}}$.

On a donc : $P(X_n - (z + (2p-1)n) \leq -nt) \leq e^{-\frac{nt^2}{2}}$. Pour $t = \frac{2p-1}{2}$ $P(X_n \leq -n\frac{2p-1}{2} + (z + (2p-1)n)) \leq e^{-\frac{n\left(\frac{2p-1}{2}\right)^2}{2}}$. Ainsi,

$$P(X_n \leq n\frac{2p-1}{2} + z) \leq e^{-n\frac{1}{8}(2p-1)^2} \tag{7.1}$$

Montrons maintenant que $\mathbb{E}(R'_h) \leq \frac{1}{e^{\frac{1}{8}(2p-1)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$:

$$\mathbb{E}(R'_h) = \sum_{n=1}^{h} \mathbb{E}(\textit{reconstruction au pas } n)$$

$$= \sum_{n=1}^{h} q\mathbb{E}(\textit{reconstruction au pas } n | \textit{on fait un pas sur la gauche et } X_n \geq 1)$$

$$= \sum_{n=1}^{h} q\mathbb{E}\left(\frac{1}{X_n}\mathbb{1}_{X_n \geq 1}\right) \text{ avec } \mathbb{1}_{X_n \geq 1} \text{ qui vaut 1 si } X_n \geq 1 \text{ et 0 sinon.}$$

Or,

$$\frac{1}{X_n}\mathbb{1}_{X_n \geq 1} = \frac{1}{X_n}\mathbb{1}_{1 \leq X_n \leq n\frac{2p-1}{2}+z} + \frac{1}{X_n}\mathbb{1}_{X_n > n\frac{2p-1}{2}+z}$$

$$\Rightarrow \frac{1}{X_n}\mathbb{1}_{X_n \geq 1} \leq \frac{1}{X_n}\mathbb{1}_{1 \leq X_n \leq n\frac{2p-1}{2}+z} + \frac{1}{n\frac{2p-1}{2}+z}$$

$$\Rightarrow \frac{1}{X_n}\mathbb{1}_{X_n \geq 1} \leq \mathbb{1}_{1 \leq X_n \leq n\frac{2p-1}{2}+z} + \frac{1}{n\frac{2p-1}{2}+z}$$

donc,

$$\mathbb{E}\left(\frac{1}{X_n}\mathbb{1}_{X_n \geq 1}\right) \leq P\left(X_n \leq n\frac{2p-1}{2} + z\right) + \frac{1}{n\frac{2p-1}{2}+z}$$

$$\Leftrightarrow q\mathbb{E}\left(\frac{1}{X_n}\mathbb{1}_{X_n \geq 1}\right) \leq qP\left(X_n \leq n\frac{2p-1}{2} + z\right) + q\frac{1}{n\frac{2p-1}{2}+z}$$

en utilisant l'équation 7.1, on obtient :

$$q\mathbb{E}\left(\frac{1}{X_n}\mathbb{1}_{X_n \geq 1}\right) \leq qe^{-n\frac{1}{8}(2p-1)^2} + q\frac{1}{n\frac{2p-1}{2}+z}$$

et ainsi :

$$\mathbb{E}(R'_h) \leq q \sum_{n=1}^{h} e^{-n\frac{1}{8}(2p-1)^2} + q \sum_{n=1}^{h} \frac{1}{n\frac{2p-1}{2} + z}$$

$\sum_{n=1}^{h} e^{-n\frac{1}{8}(2p-1)^2}$ étant une série croissante qui converge vers $\frac{1}{e^{\frac{1}{8}(2p-1)^2}-1}$, et comme $q < 1$ et $z > 1$, nous obtenons $\mathbb{E}(R'_h) \leq \frac{1}{e^{\frac{1}{8}(2p-1)^2}-1} + q\frac{2}{2p-1}\sum_{n=1}^{h}\frac{1}{n}$. Comme $\mathbb{E}(R_h) \leq \mathbb{E}(R'_h)$, le lemme est prouvé. ■

Lorsque $p = q = \frac{1}{2}$, $\mathbb{E}(z,h) \leq 2^{z-1}\sum_{i=1}^{h+z-1}\frac{1}{i}$

Dans cette partie, nous montrons que lorsque la probabilité d'avoir une requête de retrait d'un membre du groupe est égale à celle d'avoir une requête d'ajout d'un nouveau membre, l'espérance du nombre de reconstructions croit au plus logarithmiquement en nombre de requêtes. Pour cela, nous effectuons un parallèle entre une série de requêtes et un chemin de requêtes.

Définition 49 (Chemin de requêtes) *Soient z, x et $n > 0$ des entiers. Le chemin $C = (S_0, S_1, \ldots, S_n)$ reliant le point $(0, z)$ au point (n, x) est une ligne polygonale telle que ses sommets ont pour abscisse $(0, 1, \ldots, n)$ et ordonnées $(S_0, S_1, \ldots, S_n)$, avec $S_0 = z$ et $S_n = x$, et satisfaisant $S_{m+1} = S_m \pm 1$ $\forall m$.*

Soit $r_1, r_2, \ldots, r_h$ une suite de h requêtes. On remarque que pour toute suite de requêtes, il existe un et un seul chemin $C = (S_0, S_1, \ldots, S_h)$, avec $S_0 = z$ telle que $S_{m+1} = S_m + 1$ si r_{m+1} est une requête d'ajout et $S_{m+1} = S_m - 1$ si r_{m+1} est une requête de retrait. Ainsi, l'ordonnée du sommet d'abscisse h dans C correspond à la taille du groupe au bout de ces h requêtes.

Soit $C^h_{z,y}$ l'ensemble des chemins partant du point $(0, z)$ et arrivant au point (h, y). On remarque que seuls les chemins ne passant jamais par l'axe des abscisses correspondent à une suite de requêtes valides, sinon cela signifie que le groupe de communication est mort car sa taille à atteint 0.

Lemme 50 (Nombre de chemins toujours positifs) *Le nombre de chemins partant du point $(0, z)$ et arrivant au point (n, y) sans jamais toucher ou croiser l'axe des abscisses est : $\binom{n}{\frac{n+y-z}{2}} - \binom{n}{\frac{n+y+z}{2}}$.*

Preuve. Notons N_a le nombre de pas $(+1, +1)$ et N_r le nombre de pas $(+1, -1)$. Il y a eu exactement $n = N_a + N_r$ pas et nécessairement, on a $y = z + N_a - N_r$ et $N_a = \frac{y-z+n}{2}$, car $N_a = y-z+N_r = y-z+n-N_a$. Les N_a pas $(+1, +1)$ peuvent être choisis parmi les n pas possibles. Ainsi, on obtient que le nombre de chemins reliant le point $(0, z)$ au point (n, y) est $\binom{N_a+N_r}{N_a} = \binom{n}{\frac{n+y-z}{2}}$. Le principe de réflexion (voir [Fel71]) nous indique que le nombre de chemins reliant le point $(0, z)$ au point (n, y) et qui passent par l'axe des abscisses est égal au nombre de chemins reliant le point $(0, -z)$ au point (n, y). Ainsi, nous obtenons que le nombre de chemins partant du point $(0, z)$ et arrivant au point (n, y) sans jamais toucher l'axe des abscisses est $\binom{n}{\frac{n+y-z}{2}} - \binom{n}{\frac{n+y+z}{2}}$. ■

Ainsi, nous obtenons par le lemme 50 que le nombre de suites de $i-1$ requêtes qui, partant d'un groupe de taille z, terminent avec un groupe de taille x sans jamais passer par un groupe de taille 0 est $\binom{i-1}{\frac{i-1+x-z}{2}} - \binom{i-1}{\frac{i-1+x+z}{2}}$.

Lemme 51 $\binom{n}{k} - \binom{n}{k+1} = \binom{n}{k}\frac{1-n+2k}{k+1}$

Preuve. $\binom{n}{k}-\binom{n}{k+1} = \frac{n!}{k!(n-k)!}-\frac{n!}{(k+1)!(n-k-1)!} = \frac{n!(k+1)!(n-k-1)!-n!k!(n-k)!}{k!(n-k)!(k+1)!(n-k-1)!} = \binom{n}{k}\frac{(k+1)!(n-k-1)!-k!(n-k)!}{(k+1)!(n-k-1)!}$
$= \binom{n}{k}\frac{(k+1)!-k!(n-k)}{(k+1)!} = \binom{n}{k}\frac{(k+1)-(n-k)}{(k+1)} = \binom{n}{k}\frac{1-n+2k}{k+1}$ ■

Lemme 52 *Lorsque* $p = q = \frac{1}{2}$, $\mathbb{E}(1,h) \leq \sum_{i=1}^{h}\frac{1}{i}$

Preuve. Soit R^h le nombre de reconstructions après h requêtes en partant d'un groupe de taille 1 et $\mathbb{E}(R^h)$ l'espérance de R^h. L'espérance du nombre de reconstructions après h requêtes est égale à la somme des espérances du nombre de reconstructions pour chacune de ces h requêtes. Ainsi, en notant R_i le nombre de reconstructions induit par la $i^{\text{ème}}$ requête, on obtient que $\mathbb{E}(R^h) = \sum_{i=1}^{h}\mathbb{E}(R_i)$. Soit X_n la taille du groupe après n requêtes. Notons V_i l'ensemble des tailles de groupes atteignables après i requêtes en partant d'un groupe de taille initiale 1. Nous avons

$$\mathbb{E}(R_i) = \sum_{x\in V_{i-1}} \mathbb{E}(R_i|X_{i-1}=x)P(X_{i-1}=x)$$

Soit Q_n l'événement suivant : $\forall i$, avec $0 \leq i \leq n$ on a $X_i > 0$ et $\overline{Q_n}$ l'événement contraire : $\exists i$, avec $0 \leq i \leq n$ tel que $X_i \leq 0$.

Nous avons

$$\begin{aligned}\mathbb{E}(R_i) &= \sum_{x\in V_{i-1}} (\mathbb{E}(R_i|X_{i-1}=x \textit{ et } Q_{i-1})P(X_{i-1}=x \textit{ et } Q_{i-1}) \\ &+ \mathbb{E}(R_i|X_{i-1}=x \textit{ et } \overline{Q_{i-1}})P(X_{i-1}=x \textit{ et } \overline{Q_{i-1}})\big)\end{aligned}$$

Lorsque l'événement $\overline{Q_{i-1}}$ se produit, cela signifie que le groupe est mort avant la $i^{\text{ème}}$ requête et l'algorithme n'effectuera pas de reconstruction lors de la $i^{\text{ème}}$ requête car elle sera ignorée, d'où $\mathbb{E}(R_i|X_{i-1}=x \textit{ et } \overline{Q_{i-1}}) = 0$. Nous obtenons que

$$\mathbb{E}(R_i) = \sum_{x\in V_{i-1}} \mathbb{E}(R_i|X_{i-1}=x \textit{ et } Q_{i-1})P(X_{i-1}=x \textit{ et } Q_{i-1})$$

Soit C l'ensemble de tous les chemins partant du point $(0,1)$ et menant au point $(i-1,x)$ sans jamais croiser l'axe des abscisses.

$$\text{L'événement } \{X_{i-1} \textit{ et } Q_{i-1}\} \text{ est égal à } \bigcup_{c\in C}\{X_0,\ldots,X_{i-1} \text{ correspond à } c\}$$

$$P(X_{i-1}=x \textit{ et } Q_{i-1}) = \sum_{c\in C} P(X_0,\ldots,X_{i-1} \text{ correspond à } c) = |C|\left(\frac{1}{2}\right)^{i-1}, \text{ car } p=q=\frac{1}{2}$$

Par le lemme 50 et en partant d'un groupe initial de taille $z=1$, nous obtenons

$$P(X_{i-1}=x \textit{ et } Q_{i-1}) = \left(\binom{i-1}{\frac{i+x-2}{2}} - \binom{i-1}{\frac{i+x}{2}}\right)\left(\frac{1}{2}\right)^{i-1}$$

Par définition, $\mathbb{E}(R_i|X_{i-1} = x \text{ et } Q_{i-1})$ est égal à la probabilité, pour un groupe de taille x, d'effectuer une reconstruction lors de la réception de la $i^{\text{ème}}$ requête (c'est-à-dire la suivante), multipliée par le coût de cette reconstruction (c'est-à-dire 1). Nous obtenons que $\mathbb{E}(R_i|X_{i-1} = x \text{ et } Q_{i-1}) = \frac{1}{2x}$. Ainsi,

$$\mathbb{E}(R_i) = \sum_{x \in V_{i-1}} \frac{1}{2x} \left(\binom{i-1}{\frac{i+x-2}{2}} - \binom{i-1}{\frac{i+x}{2}} \right) \left(\frac{1}{2}\right)^{i-1}$$

et par application du lemme 51, on obtient que

$$\mathbb{E}(R_i) = \sum_{x \in V_{i-1}} \frac{1}{i+x} \binom{i-1}{\frac{i+x-2}{2}} \left(\frac{1}{2}\right)^{i-1}$$

Nous en déduisons que

$$\mathbb{E}(R_i) \leq \frac{1}{i} \left(\frac{1}{2}\right)^{i-1} \sum_{x \in V_{i-1}} \binom{i-1}{\frac{i+x-2}{2}}$$

et comme

$$\sum_{x \in V_{i-1}} \binom{i-1}{\frac{i+x-2}{2}} \leq \sum_{j=1}^{i-1} \binom{i-1}{j} \leq 2^{i-1}$$

nous obtenons que $\mathbb{E}(R_i) \leq \frac{1}{i}$ et donc que $\mathbb{E}(1,h) \leq \sum_{i=1}^{h} \frac{1}{i}$, ce qui prouve le résultat. ■

Lemme 53 *Lorsque* $p = q = \frac{1}{2}$, $\mathbb{E}(z,h) \leq 2^{z-1}\mathbb{E}(1,h+z-1)$.

Preuve. Soit $S = r_1, \ldots, r_h$ une suite de h requêtes. La requête r_i concerne l'ajout d'un nouveau membre avec probabilité p et concerne le retrait d'un membre avec probabilité $q = 1-p$. Soit R_h^z le nombre de reconstructions induites par les h premières requêtes de S en partant d'un groupe de taille initiale z. On remarque que

$$\mathbb{E}(R_h^z) = \mathbb{E}(R_h^z|r_1 \textit{ est un ajout})P(r_1 \textit{ est un ajout}) + \mathbb{E}(R_h^z|r_1 \textit{ est un retrait})P(r_1 \textit{ est un retrait})$$

et donc que

$$\mathbb{E}(R_h^z) \geq \mathbb{E}(R_h^z|r_1 \textit{ est un ajout})P(r_1 \textit{ est un ajout})$$

Si r_1 est une requête d'ajout, cela signifie que notre algorithme n'effectue pas de reconstruction. Ainsi, le nombre de reconstructions induit par S est égal au nombre de reconstructions induit par $S - r_1$ en partant d'un groupe de taille $z+1$. Nous avons donc $\mathbb{E}(R_h^z|r_1 \textit{ est un ajout})P(r_1 \textit{ est un ajout}) = \mathbb{E}(R_{h-1}^{z+1})p$. Ainsi, nous obtenons que $\mathbb{E}(R_h^z) \geq \mathbb{E}(R_{h-1}^{z+1})p$ et donc que $\frac{1}{p}\mathbb{E}(R_h^z) \geq \mathbb{E}(R_{h-1}^{z+1})$. Ainsi, nous avons :

$$\mathbb{E}(z,h) \leq \frac{1}{p}\mathbb{E}(z-1,h+1) \leq \frac{1}{p^2}\mathbb{E}(z-2,h+2) \leq \ldots \leq \frac{1}{p^{z-1}}\mathbb{E}(1,h+z-1)$$

et comme $p = \frac{1}{2}$, nous obtenons le résultat. ■

7.4 Conclusion, perspectives

Dans ce chapitre nous avons proposé des résultats analytiques sur le nombre de fois où il faut reconstruire un arbre de connexion pour qu'il garde un diamètre proche de l'optimal tout en supportant les adaptations nécessaires dues aux entrées et sorties de membres dans le groupe évolutif qu'il connecte. Nous avons montré que dans le pire cas *tout algorithme* (respectant les contraintes du problème) doit reconstruire un nombre de fois linéaire en le nombre de requêtes. Ce résultat est donc une limite intrinsèque au problème. Les résultats analytiques en moyenne que nous proposons ensuite constituent donc une bonne nouvelle puisque nous montrons (grâce à une modélisation par des marches aléatoires) que lorsque le groupe initial est petit, la moyenne du nombre de reconstructions est au plus logarithmique en le nombre de requêtes. Notre procédé de mise-à-jour présente des caractéristiques intéressantes en termes de possibilités pour être réparti car la décision de reconstruire est locale et que les ajouts/retraits des membres qui ne sont pas la racine se font facilement.

Le travail fait dans ce chapitre pourrait être étendu à d'autres critères, comme le poids des arbres (pouvant représenter leurs coûts) ou la *distance moyenne* dans l'arbre pour affiner le contrôle des distances entre les membres. Des études en pires cas ont déjà été menées sur ce type de paramètres dans [Thi06, TL07b, TL04, TL06a, TL07a]. Une analyse en moyenne de ces résultats devra être envisagée pour affiner notre connaissance des mécanismes de mise-à-jour de ces structures de connexion qui peuvent être le supports de multiples types d'applications : connexions d'internautes mais aussi connexion de machines dans une grille pour des calculs avec des machines à ressources spécifiques et changeantes, etc.

De manière plus large nous pensons que les techniques et résultats dans le domaine de l'algorithmique d'approximation, l'algorithmique on-line peuvent aider à contrôler/maîtriser certains paramètres de structures discrètes qui apparaissent (de manière parfois purement implicite) dans les travaux sur les services répartis. On peut penser à la construction d'arbres comme dans le présent chapitre mais aussi à la construction de points de contrôle dans un réseau qui peuvent faire appel à des notions de dominants (éventuellement multiples pour tolérer des pannes) ou de couvertures par exemple. Ces problèmes sont étudiés depuis longtemps en théorie et algorithmique des graphes. La limitation du cadre restrictif de l'exécution purement off-line (nécessité d'avoir la connaissance intégrale des données) commence à être levé avec des modèles alternatifs de type on-line, particulièrement adaptés aux environnements changeants caractéristiques des systèmes répartis. Cependant, ces travaux ne tiennent en général pas compte d'exécutions réparties et évaluent la qualité des méthodes en pire cas. Ce dernier point peut conduire à juger une méthode uniquement dans la pire des situations, ce qui n'est pas forcément très représentatif de son fonctionnement courant. Une analyse plus fine (en moyenne par exemple comme dans ce chapitre) est alors nécessaire pour les comparer. Cette analyse peut alors être délicate et non triviale et faire appel à des résultats de type probabiliste. Un rapprochement (déjà en marche dans certains centres de recherche, notamment en France) des diverses communautés thématiques en informatique est maintenant indispensable pour maîtriser le comportement de ces systèmes. Ceux-ci ont une importance pratique, un intérêt théorique et doivent être compris en profondeur pour pouvoir être déployés et exploités au mieux.

Conclusion générale

Dans cette thèse, nous avons montré que le rapport d'approximation en pire cas, qui constitue un outil précieux dans l'analyse des algorithmes, ne reflète pas la réalité de toutes les exécutions et qu'il n'est pas suffisant pour juger *a priori* des performances d'un algorithme. Pourtant, conformément au besoin de garanties absolues sur la qualité des solutions obtenues, c'est ce critère d'évaluation qui est généralement retenu dans la littérature. On peut légitimement se demander comment mieux évaluer les performances d'un algorithme. Cette question n'est pas nouvelle et différentes mesures ont déjà été proposées comme le rapport d'approximation différentiel ou bien l'espérance de la taille des solutions retournées par exemple. Toutes ces mesures reflètent notre désir de capturer le comportement des algorithmes afin de les utiliser au mieux. Malheureusement, il n'existe pas, *a priori*, de mesure « ultime » permettant de classer les algorithmes les uns par rapport aux autres*. Nous avons étudié le comportement de différents algorithmes en prenant en compte l'indéterminisme, présent dans de nombreux algorithmes ou dans l'absence de connaissance du futur, qui peut influencer très fortement (en bien comme en mal) leurs performances. Cela nous a permis d'une part de dépasser certaines limites intrinsèques à un problème (comme dans la seconde partie, sur les groupes dynamiques, où nous avons montré que malgré un mauvais comportement en pire cas pour tout algorithme, le comportement moyen d'un algorithme peut être bon, voire excellent), et d'autre part, dans la partie sur le vertex cover, de nous rendre compte que les algorithmes qui possèdent le meilleur rapport d'approximation en pire cas sont ceux qui, finalement, possèdent les plus mauvaises performances en moyenne et qu'au contraire, les algorithmes qui se comportent mal sur un nombre très limité d'instances (au regard du nombre total d'instances possibles) possèdent un comportement excellent en moyenne. De plus, nous avons exhibé un algorithme qui, malgré un très mauvais comportement en pire cas, se comporte extrêmement bien en moyenne pour toute instance (voire chapitre 3) et nous avons montré qu'il se comporte toujours mieux qu'un autre algorithme (voire chapitre 2), montrant ainsi que des relations de dominances non triviales peuvent apparaître. Nous avons aussi proposé une mesure (dérivée du rapport d'approximation différentiel) qui capture la qualité moyenne d'un algorithme sur une instance donnée.

Afin de poursuivre ces travaux sur la comparaison des performances et du comportement des algorithmes, il serait intéressant de déterminer des relations de dominance sur des classes de graphes plus ou moins importantes comme nous l'avons fait sur les chemins. Prouver la conjecture annoncée dans le chapitre 5 constituerait une avancée significative dans ce domaine. Comme nous l'avons vu, l'évaluation en moyenne permet de contourner certaines difficultés. Par exemple, on ne connaît toujours pas d'algorithme pour le problème de l'ensemble indépendant de taille maximum dont le

*Plus précisément, les performances des algorithmes n'admettent pas d'ordre total, que ce soit pour la taille des solutions ou pour la taille moyenne. Pour s'en convaincre, il suffit de voir le comportement des algorithmes Edge Deletion et Maximum Degree Greedy dans le chapitre 1.

rapport d'approximation en pire cas soit borné par une constante, bien que ce problème et celui du vertex cover soient intimement liés. L'exploration du comportement moyen d'algorithme pour ce problème permettrait sûrement d'y arriver.

Pour conclure, je dirai que l'évaluation en moyenne doit se généraliser autant que possible car elle prend en compte l'ensemble des exécutions possibles d'un algorithme et elle ne doit évidemment pas remplacer l'évaluation en pire cas, mais la compléter.

Bibliographie

[AA92] Noga Alon and Yossi Azar. On-line steiner trees in the euclidean plane. In *SCG '92 : Proceedings of the eighth annual symposium on Computational geometry*, pages 337–343, New York, NY, USA, 1992. ACM.

[AAA+06] Noga Alon, Baruch Awerbuch, Yossi Azar, Niv Buchbinder, and Joseph (Seffi) Naor. A general approach to online network optimization problems. *ACM Transactions on Algorithms*, 2(4) :640–660, 2006.

[AAB96] Baruch Awerbuch, Yossi Azar, and Yair Bartal. On-line generalized steiner problem. In *SODA '96 : Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 68–74, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[AGN01] Jochen Alber, Jens Gramm, and Rolf Niedermeier. Faster exact algorithms for hard problems : a parameterized point of view. *Discrete Mathematics*, 229(1-3) :3–27, 2001.

[AI07] David Avis and Tomokazu Imamura. A list heuristic for vertex cover. *Operations Research Letters*, 35(2) :201–204, 2007.

[AIS03] Frank Adelstein, Golden G. Richard III, and Loren Schwiebert. Distributed multicast tree generation with dynamic group membership. *Computer communications*, 26 :1105–1128, 2003.

[AKCF+04] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Chris T. Symons. Kernelization algorithms for the vertex cover problem : Theory and experiments (extended abstract). In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 62–69, New Orleans, Louisiana, USA, 2004.

[Ang07] Spyros Angelopoulos. Improved bounds for the online steiner tree problem in graphs of bounded edge-asymmetry. In *SODA '07 : Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 248–257, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[APMS+99] Giorgio Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation : Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[AS64] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition, 1964.

[BBL06] Fabien Baille, Lelia Blin, and Christian Laforest. Distributed approximation allocation ressources algorithm for connecting groups. In *Proceedings of the European Conference on Parallel Computing*, pages 519–529. LNCS 4128, 2006.

[BDL09] Etienne Birmelé, François Delbot, and Christian Laforest. Mean analysis of an online algorithm for the vertex cover problem. *Information Processing Letters*, 109(9) :436–439, 2009.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998.

[BFR98] R. Balasubramanian, Michael R. Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Information Processing Letters*, 65(3) :163–168, 1998.

[Bol01] B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.

[Bre97] Pierre Bremaud. *Introduction aux probabilités*. Springer, 1997.

[BYE85] R. Bar-Yehuda and S. Even. A local ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25, 1985.

[CDRC+03] James Cheetham, Frank Dehne, Andrew Rau-Chaplin, Ulrike Stege, and Peter J. Taillon. Solving large fpt problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, 67(4) :691–706, 2003.

[CKJ01] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover : further observations and further improvements. *Journal of Algorithms*, 41(2) :280–301, 2001.

[CLL+05] J. Cardinal, M. Labbé, S. Langerman, E. Levy, and H. Mélot. A tight analysis of the maximal matching heuristic. In *Computing and Combinatorics : 11th Annual International Conference, COCOON 2005*, volume 3595 of *Lecture Notes in Computer Science*, pages 701 – 709, Kunming, China, 2005. Springer–Verlag.

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction à l'algorithmique*. Dunod, 1990.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[Del06] François Delbot. Rapport de stage de M2 recherche. Comparaison de deux algorithmes d'approximation pour le problème de la couverture d'arêtes par des sommets. Master's thesis, Laboratoire IBISC. Université d'Evry val d'Essonne, 2006.

[DF95] Rod G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II : On completeness for $W[1]$. *Theoretical Computer Science*, 141(1–2) :109–131, 1995.

[DF98] R. G. Downey and M. R. Fellows. Parameterized complexity. *Springer-Verlag Heidelberg*, 1998.

[DFP93] Martin Dyer, Alan Frieze, and Boris Pittel. The average performance of the greedy matching algorithm. *Annals of Applied Probability*, 1993.

[DFS97] Rodney G. Downey, Michael R. Fellows, and Ulrike Stege. Parameterized complexity : A framework for systematically confronting computational intractability. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99, 1997.

[DL08] François Delbot and Christian Laforest. A better list heuristic for vertex cover. *Information Processing Letters*, 107(3-4) :125–127, 2008.

[DLT07] François Delbot, Christian Laforest, and Nicolas Thibault. Evaluation en moyenne d'un algorithme pour la mise à jour d'arbre de connexion. In *AlgoTel 2007, 9ème rencontres francophones sur les aspects algorithmiques de télécommunications*, 2007.

[DP05] Marc Demange and Vangelis Th. Paschos. On-line vertex-covering. *Theoretical Computer Science*, 332(1-3) :83–108, 2005.

[Fel71] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. Wiley, 1971.

[Fel01] Michael R. Fellows. Parameterized complexity : The main ideas and some research frontiers. In *ISAAC '01 : Proceedings of the 12th International Symposium on Algorithms and Computation*, pages 291–307, London, UK, 2001. Springer-Verlag.

[FW98] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998.

[Gau99] Bruno Gauthier. *Calcul symbolique sur les séries hypergéométriques*. PhD thesis, Université de Marne-la-Vallée, 1999.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability*. Freeman and Co., New York, 1979.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *STOC '74 : Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, New York, NY, USA, 1974. ACM Press.

[GM02] A. Goel and K. Munagala. Extending greedy multicast routing to delay sensitive applications. *Algorithmica*, 33(3) :335–352, 2002.

[HKP98] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *SODA '98 : Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 219–225, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[Hoc97] Dorit S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Compagny, Boston, MA, USA, 1997.

[II86] Amos Israel and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2) :77–80, 1986.

[IW91] Makoto Imase and Bernard Waxman. Dynamic steiner tree problem. *SIAM Journal of Discrete Mathematics*, 4(3) :369–384, 1991.

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar05] George Karakostas. A better approximation ratio for the vertex cover problem. In *International Colloquium on Automata, Languages and Programming*, pages 1043–1050, 2005.

[KB94] Sami Khuri and Thomas Bäck. An evolutionary heuristic for the minimum vertex cover problem. In Jörn Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation – Proc. of the KI-94 Workshop*, pages 86–90, Saarbrücken, Germany, 1994.

[KG03] Ketan Kotecha and Nilesh Gambhava. A hybrid genetic algorithm for minimum vertex cover problem. In Bhanu Prasad, editor, *1st Indian International Conference on Artificial Intelligence*, pages 904–913, Hyderabad, December 2003. IICAI.

[KMW04] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally ! In *PODC '04 : Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 300–309, New York, NY, USA, 2004. ACM.

[KMW06] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *SODA '06 : Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 980–989, New York, NY, USA, 2006. ACM.

[KR08] 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.

[LC05] Li Layuan and Li Chunlin. A qos multicast routing protocol for dynamic group topology. *Information Sciences : an International Journal*, 169(1-2) :113–130, 2005.

[McD98] C. McDiarmid. Concentration. In M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed (Eds.), editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, pages 195–248. Springer, 1998.

[MPT03] Jérôme Monnot, Vangelis T. Paschos, and Sophie Toulouse. *Polynomial approximation of hard NP-problems : local extremals and differential relation. (Approximation polynomiale des problèmes NP-difficiles : optima locaux et rapport différentiel.)*. Hermes Science Publications, 2003.

[MS85] Burkhard Monien and Ewald Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22(1) :115–123, 1985.

[NR00] Rolf Niedermeier and Peter Rossmanith. On efficient fixed parameter algorithms for weighted vertex cover. In *International Symposium on Algorithms and Computation*, pages 180–191, 2000.

[Pas97] Vangelis T. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys (CSUR)*, 29(2) :171–209, 1997.

[PR01] Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2) :97–100, 2001.

[PS09] Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12) :642–645, 2009.

[PY88] Christos Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. In *STOC '88 : Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 229–234, New York, NY, USA, 1988. ACM Press.

[RK00] Chantal Roth-Korostensky. *Algorithms for building multiple sequence alignments and evolutionary trees*. PhD thesis, ETH Zürich, Institute of Scientific Computing, 2000.

[Sav82] Carla Savage. Depth-first search and the vertex cover problem. *Information Processing Letters*, 14(5) :233–235, July 1982.

[SJSL04] Peng-Yeng Yin Shyong Jian Shyu and Bertrand M.T. Lin. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research*, pages 283–304, 2004.

[SMM99] R. Sriram, G. Manimaran, and C. Murthy. A rearrangeable algorithm for the construction of delay-constrained dynamic multicast trees. *IEEE-ACM Transactions on Networking*, 7 :514–529, 1999.

[Ste00] Ulrike Stege. *Resolving conflicts from problems in computational biology.* PhD thesis, ETH Zürich, Institute of Scientific Computing, 2000.

[Thi06] Nicolas Thibault. *Algorithmes d'approximation pour l'optimisation en ligne d'ordonnancements et de structures de communications.* PhD thesis, Université d'Evry-Val d'Essonne, 2006.

[TL04] Nicolas Thibault and Christian Laforest. An optimal online strategy to increment connection trees. In *Workshop* Adaptive Wireless Networks *of Globecom*, 2004.

[TL06a] Nicolas Thibault and Christian Laforest. Ajouts et retraits dans un arbre de connexion. In *AlgoTel*, pages 33–36, 2006.

[TL06b] Nicolas Thibault and Christian Laforest. An optimal rebuilding strategy for a decremental problem. In *International colloquium on structural information and communication complexity (SIROCCO)*, volume LNCS 4056, pages 157–170, 2006.

[TL07a] Nicolas Thibault and Christian Laforest. Minimizing the number of critical stages for the on-line steiner tree problem. In *International Network Optimization Conference*, 2007.

[TL07b] Nicolas Thibault and Christian Laforest. An optimal rebuilding strategy for an incremental tree problem. *Journal of Interconnection Networks*, Vol. 8 :75–99, 2007.

[Wax88] Bernard Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9) :1617–1622, 1988.

[Wil06] Herbert S. Wilf. *Generatingfunctionology.* A. K. Peters, Ltd., Natick, MA, USA, 2006.

[XBHL07] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction : Easy generation of hard (satisfiable) instances. *Artif. Intell.*, 171(8-9) :514–534, 2007.

[XCW04] Xinshun Xu Songsong Li Guangpu Xia Xiaoming Chen, Zheng Tang and Jiahai Wang. *Lecture Notes in Computer Science*, volume 3173, chapter An Algorithm Based on Hopfield Network Learning for Minimum Vertex Cover Problem, pages 430–435. 2004.

[XL06] Ke Xu and Wei Li. Many hard examples in exact phase transitions. *Theor. Comput. Sci.*, 355(3) :291–302, 2006.

[XM06] Xinshun Xu and Jun Ma. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing*, 69(7-9) :913–916, 2006.

[YK98] Shih-Yi Yuan and Sy-Yen Kuo. A new technique for optimization problems in graph theory. *IEEE Transactions on Computers*, 47(2) :190–196, 1998.

[ZGF+06] Yong Zhang, Qi Ge, Rudolf Fleischer, Tao Jiang, and Hong Zhu. Approximating the minimum weight weak vertex cover. *Theoretical Computer Science*, 363(1) :99–105, 2006.

[ZM06] Fang Zhao and Muriel Médard. Online network coding for the dynamic multicast problem. In *IEEE International Symposium on Information Theory*, pages 1753–1757, 2006.