PROJET UE COMPLEX — MU4IN900
COUVERTURE DE GRAPHE

ARIANA CARNIELLI

1. Introduction

Ce projet s'intéresse au problème de couverture de graphe (*vertex cover*). Étant donné un graphe $G = (V, E)$, où V est l'ensemble de ses n sommets et E l'ensemble de ses m arêtes, une *couverture* de G est un sous-ensemble de sommets $V' \subseteq V$ tel que toute arête $e \in E$ a au moins une de ses extrémités dans V'. Le problème de couverture de graphe consiste à trouver une couverture avec le nombre minimal de sommets. Ce problème est NP-difficile.

On test dans ce projet deux types d'algorithmes : approchés, calculant une solution approchée en temps polynomial en termes de n et m, décrits dans la Section 3, et exacts, calculant une solution en temps exponentiel, décrits dans la Section 4.

2. Implémentation

Dans cette section, on présente les choix d'implémentation faits dans ce projet. Toute l'implémentation a été faite en langage Python, en se basant sur le module `networkx` pour représenter les graphes, `numpy` pour représenter des tableaux, `time` pour les calculs de temps d'exécution, et `matplotlib` pour l'affichage de graphiques.

Les fichiers fournis joints à ce rapport, décrits dans la suite de cette section, sont :

- Fichiers contenant les fonctions codées et les tests : `projet.py`, `fonctionsTests.py`, `Tests.ipynb`.
- Fichiers avec du code auxiliaire : `progressBar.py`.
- Documentation : `projet.html`, `fonctionsTests.html`.
- Plusieurs fichiers en format `.npz` avec les données expérimentales utilisées dans la construction des graphiques présentés dans ce rapport.

2.1. `projet.py`.

Il s'agit du fichier principal, contenant la classe `Graphe`, qui représente un graphe non-orienté. Son constructeur permet de créer un graphe vide, un graphe à partir d'un fichier ou un graphe aléatoire avec nombre de sommets et probabilité de présence d'une arête donnés en argument. Quelques méthodes de manipulation de graphes utiles dans la suite sont codées dans cette classe, ainsi que tous les algorithmes pour le problème de couverture de graphe.

2.2. `fonctionsTests.py`.

Ce fichier contient quatre fonctions permettant de réaliser des tests : une qui teste le temps d'exécution pour une probabilité de présence d'arête fixée et des tailles variables de graphe, une qui fait le même mais avec une probabilité qui dépend de la taille du graphe, une qui teste l'écart entre les réponses des algorithmes approchés et exacts, et une qui calcule le nombre de nœuds visités dans l'arbre utilisé dans les algorithmes exacts.

2.3. `Tests.ipynb`.

Ce fichier Jupyter Notebook réalise des tests à l'aide des fonctions du fichier précédent. Chaque test est réalisé dans deux cellules : une première où le test en soi est réalisé et ses résultats sont enregistrés dans un fichier au format `.npz` et une deuxième qui lit les données du fichier correspondant et en produit un graphique. Comme la durée de chaque test est assez longue, ce découpage permet de re-tracer des graphiques sans avoir besoin de re-faire le test.

2.4. **Autres fichiers.**

Le fichier `progressBar.py` permet d'afficher des barres de progression lors des tests et a été repris d'un projet précédent. Les fichiers `.html` de documentation ont été crées à partir des fichiers `.py` et de l'outil `pydoc` et permettent de comprendre plus en détail le fonctionnement de chaque fonction. Les fichiers `.npz` contiennent les données de simulation.

3. Algorithmes approchés

Deux algorithmes approchés ont été implémentés : un algorithme basé sur un couplage maximal et un algorithme glouton.

Un *couplage* de $G = (V, E)$ est un sous-ensemble d'arêtes $E' \subseteq E$ tel que deux arêtes différentes de E' n'ont pas d'extrémité en commun. Il est dit *maximal* s'il n'est pas possible de rajouter d'arêtes à E' tout en restant un couplage. On peut obtenir une couverture V' de G à partir d'un couplage E'

en prenant V' comme l'ensemble des extrémités des arêtes de E'. Comme vu en cours, il s'agit bien d'une couverture et cet algorithme est 2-approché pour le problème de la couverture de graphe.

L'algorithme glouton consiste à construire V' itérativement en y rajoutant un sommet de plus grand degré de G, supprimant ce sommet de G (et toutes les arêtes qui y sont incidentes), et continuant la procédure jusqu'à ce que G n'ait plus d'arêtes. Deux versions de cet algorithme ont été implémentées : dans une première version, à chaque fois que l'on supprime un sommet de G, on crée une nouvelle copie de G. Les tests ont montré que ces multiples copies ont un très grand impact sur le temps d'exécution. Ainsi, on a décidé d'en faire une nouvelle version où une seule copie est faite au début et modifiée itérativement.

3.1. **Résultats théoriques.**

Proposition. *L'algorithme glouton n'est pas optimal.*

Démonstration. On considère le graphe de la figure ci-contre. Tous les sommets ont le même degré, donc l'algorithme glouton peut choisir n'importe lequel. Supposons qu'il choisit a. Après avoir supprimé a, les sommets b et f ont degré 1 et les autres ont encore degré 2. L'algorithme glouton pourrait alors choisir le sommet d. Après ce choix, les quatre sommets restants ont degré 1 et il reste les arêtes (b,c) et (e,f). L'algorithme est donc obligé de prendre deux autres sommets avant de se finir, ce qui fera une solution avec 4 sommets, par exemple, $\{a,b,d,f\}$. Or, $\{a,c,e\}$ est une couverture de ce graphe avec moins de sommets, donc la solution retournée par l'algorithme glouton n'est pas optimale. □

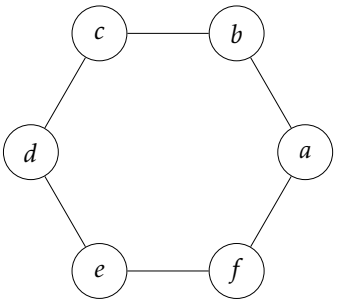


FIGURE 1. Graphe utilisé dans la preuve.

Proposition. *L'algorithme glouton n'est pas r-approché pour aucun r.*

Démonstration. Soit k un entier positif arbitraire. Soit $G = (V,E)$ le graphe construit de la façon suivante : l'ensemble des sommets se décompose comme $V = A \cup B$, où A et B sont deux sous-ensembles disjoints et G est biparti selon A et B. L'ensemble A contient k sommets. L'ensemble B se décompose en k sous-ensembles disjoints deux à deux, $B_1, \ldots, B_k$.

Pour chaque $i \in \{1,\ldots,k\}$, l'ensemble de sommets B_i contient $\left\lfloor \frac{k}{i} \right\rfloor$ sommets. Chaque sommet de B_i est relié à exactement i sommets de A et deux sommets de B_i n'ont pas d'arête vers un même sommet de A. Par exemple, le cas $k = 4$ est représenté dans la Figure 2.

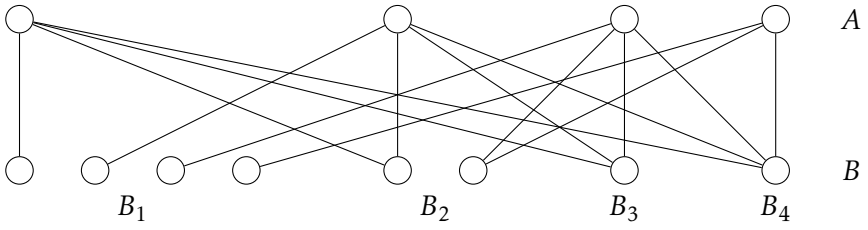


FIGURE 2. Graphe utilisé dans la preuve avec $k = 4$.

Les sommets de A ont degré au plus k, car chaque sommet de A ne peut avoir d'arête que vers au maximum un sommet de chaque B_i. Chaque sommet de B_i a degré exactement i. Le degré maximal de ce graphe est donc k, qui est atteint par l'unique sommet de B_k.

L'algorithme glouton prend toujours un sommet de degré maximal. Il peut donc commencer par prendre l'unique sommet de B_k. En le supprimant du graphe, les sommets de A auront degré au maximum $k-1$, qui est aussi le degré des sommets de B_{k-1}. Ainsi, l'algorithme glouton peut ensuite prendre les sommets de B_{k-1}. Après les avoir pris et supprimé du graphe, les sommets de A auront degré au maximum $k-2$, qui est aussi le degré des sommets de B_{k-2}. En continuant comme ci-dessus, l'algorithme pourra donc prendre B comme sa couverture. Cette couverture aura alors une quantité de sommets égale à

$$\sum_{i=1}^{k} |B_i| = \sum_{i=1}^{k} \left\lfloor \frac{k}{i} \right\rfloor = \Theta\left(\sum_{i=1}^{k} \frac{k}{i} \right) = \Theta\left(k \sum_{i=1}^{k} \frac{1}{i} \right) = \Theta(k \log k).$$

Or, comme le graphe est biparti, A est aussi une couverture possible du graphe, qui contient k sommets. La couverture optimale a donc au maximum k sommets. Supposons par l'absurde que l'algorithme glouton soit r-optimal pour un certain $r \geq 1$. En particulier, on aurait

$$|B| \leq r|\text{opt}| \qquad \text{et} \qquad |\text{opt}| \leq |A|,$$

où opt est une solution optimale du problème de couverture de graphe. En combinant ces deux inégalités, on a

$$|B| \leq r|A|.$$

Or, $|B| = \Theta(k \log k)$ et $|A| = k$, donc

$$\Theta(k \log k) \leq rk,$$
$$\Theta(\log k) \leq r.$$

Ici, r est une constante fixée mais k est un entier positif arbitraire. L'inégalité ci-dessus est donc une contradiction car $\log k \to +\infty$ lorsque $k \to +\infty$, et donc $\Theta(\log k)$ ne peut pas rester borné par la constante r, peut importe la valeur choisie pour r. Cette contradiction montre donc que l'algorithme glouton n'est pas r-approché pour aucun r. □

3.2. **Résultats expérimentaux.** On a testé le temps d'exécution des algorithmes approchés de couplage et glouton. Pour chaque algorithme, on a pris 10 valeurs de n équirépartis entre 50 et 500 et 3 valeurs de p. Pour chaque paire de valeurs de n et p, chaque algorithme a été exécuté 100 fois sur des instances aléatoires de graphes (ce nombre de 100 répétitions a aussi été utilisé dans toutes les autres simulations de temps de calcul présentées dans ce rapport). Les temps d'exécution moyens ainsi obtenus sont représentes dans la Figure 3.

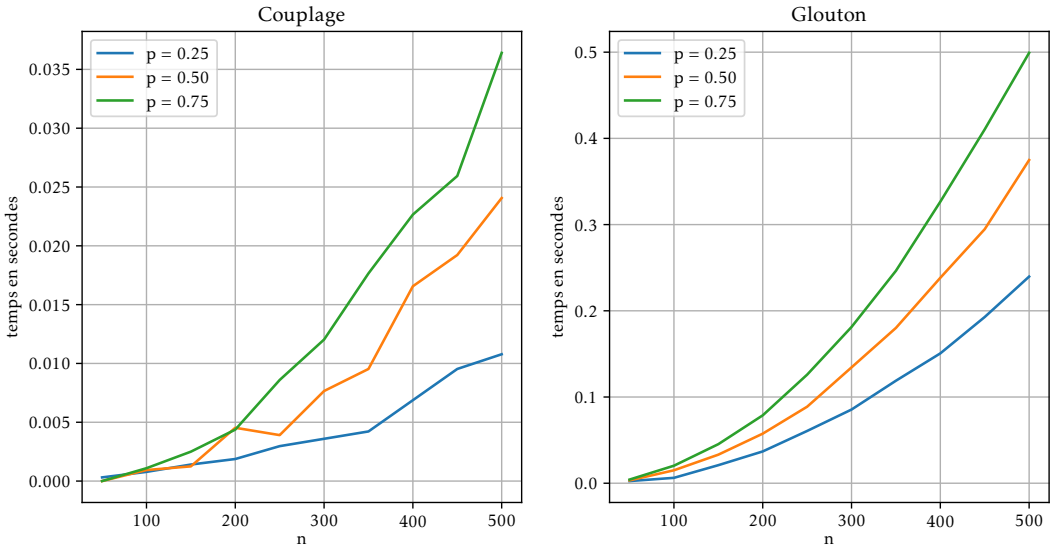


FIGURE 3. Temps de calcul en fonction de n pour les algorithmes approchés

On observe dans la Figure 3 que l'algorithme de couplage est plus performant. En regardant son code, on remarque qu'il s'exécute en $\Theta(m)$, car il s'agit d'une seule boucle sur les arêtes, le corps de la boucle étant exécuté en $\Theta(1)$. Pour l'algorithme glouton, sa boucle principale retire un sommet du graphe à chaque itération, donc elle est exécutée $O(n)$ fois. Dans le corps de la boucle, il faut calculer un sommet de degré maximal et le retirer du graphe, deux opérations qui sont faites au pire en $O(n+m)$. On a donc une complexité de $O(n(n+m))$ pour l'algorithme glouton, qui est supérieure à celle de l'algorithme de couplage.

Le nombre maximal d'arêtes dans un graphe à n sommets est $\frac{n(n-1)}{2}$. Avec une probabilité p de présence d'une arête, le nombre espéré d'arêtes dans un graphe aléatoire est $p\frac{n(n-1)}{2}$. Donc, en espérance, $m = p\frac{n(n-1)}{2} = \Theta(pn^2)$, ce qui veut dire que, en termes de n et p, l'algorithme de couplage est en $\Theta(pn^2)$ et le glouton, en $O(pn^3)$. Les courbes de la Figure 3 mettent en évidence ce caractère polynomial en n ainsi que la dépendance en p.

Pour la qualité des approximations, on a comparé les résultats de cet algorithme avec la solution optimale obtenue avec l'algorithme de branchement amélioré de la Section 4. Cet algorithme étant exponentiel, on a été contraint de limiter la taille des graphes pour ces simulations à 50 sommets au maximum.

Pour chaque algorithme approché et chaque paire de valeurs de n et p, on a crée 50 graphes aléatoires et mesuré le rapport entre le nombre de sommets dans la solution approchée sur le nombre de sommets de la solution exacte. En effet, comme, pour un n et p données, la taille de la solution optimale peut encore varier beaucoup, on a choisi de faire le rapport, et non une différence, pour limiter les effets de cette variation. La Figure 4 montre, pour chaque algorithme et chaque valeur de n et p, la médiane (cercles) et les deux quartiles (extrémités de la barre) obtenus avec les 50 graphes aléatoires. Les valeurs de n choisis ont été $5, 10, 15, \ldots, 50$ mais, pour pouvoir visualiser les points avec différentes valeurs de p sur une même image, les points pour $p = 0.25$ et $p = 0.75$ sont légèrement décalés (respectivement à gauche et à droite).

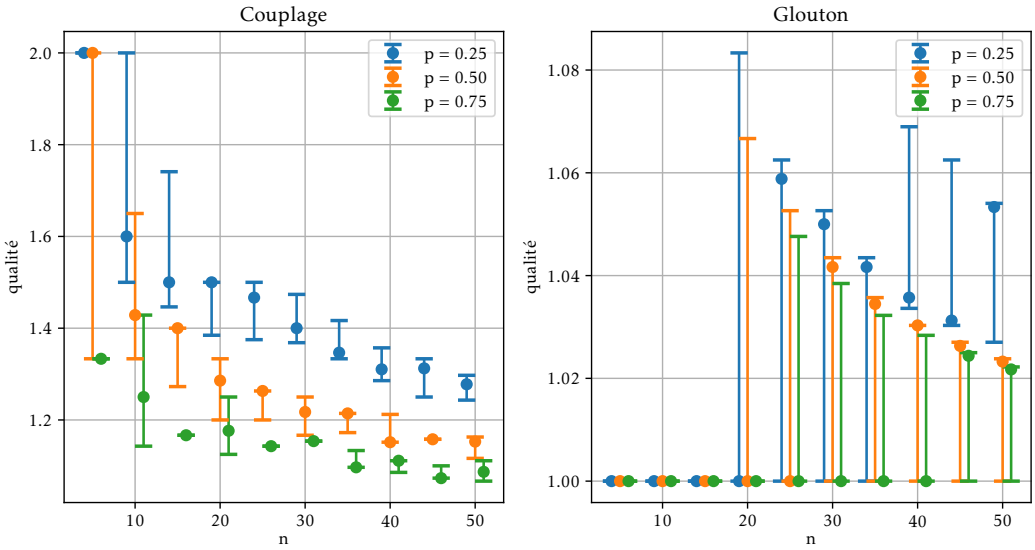


FIGURE 4. Qualité d'approximation pour les algorithmes approchés

Comme l'algorithme de couplage est 2-approché, tous les rapports calculés doivent être entre 1 et 2, ce que l'on observe sur la Figure 4. La qualité de l'approximation s'améliore lorsque n ou p augmentent ; la distance entre les deux quartiles diminue aussi, ce qui indique une plus grande concentration des rapports autour d'une valeur. Par contre, pour toutes les valeurs de n et p, tous les premiers quartiles calculés sont supérieurs à 1, ce qui veut dire que moins de 25% des solutions trouvées sont exactes (et, en regardant les positions des quartiles et de la médiane, on peut s'attendre à ce que la proportion de solutions exactes trouvées soit très inférieure à 25%). On s'attend donc à ce que l'algorithme de couplage ne donne quasiment jamais la solution exacte.

On observe au contraire que l'algorithme glouton a une très bonne performance. Pour des valeurs petites de n (jusqu'à 15), médiane et les deux quartiles sont tous égaux à 1, ce qui veut dire qu'au moins 75% des solutions trouvées sont exactes. La situation se dégrade à $n = 20$, mais ensuite l'écart diminue vers 1 lorsque n ou p augmentent. Pour $p = 0.50$ ou $p = 0.75$, le premier quartile vaut 1, ce qui veut dire qu'au moins 25% des solutions trouvées sont exactes. On remarque aussi que le pire quartile est de l'ordre de 1.08 (soit une solution 8% plus grande que la solution exacte), mais cela est aussi le meilleur quartile de l'algorithme de couplage.

Cela pourrait sembler dans un premier temps incohérent avec le résultat théorique qui garantit que l'algorithme glouton n'est pas r-approché pour aucune valeur de r. Les résultats obtenus peuvent s'expliquer par le fait que les cas où l'algorithme glouton marche très mal — comme celui décrit dans la Section 3.1 — sont très rares et très improbables d'être trouvés en pratique. Ainsi, le pire des cas de l'algorithme glouton est bien pire que le pire des cas de l'algorithme de couplage, mais la Figure 4 semble suggérer que l'algorithme glouton est meilleur dans la plupart des cas.

4. Algorithmes exacts

Les algorithmes exacts implémentés se basent sur une méthode de branchement. Le premier de ces algorithmes, implémenté à la méthode `algoBranchement`, choisit à chaque étape une arête (u, v) et considère deux branches : dans la première, il considère uniquement les couvertures contenant u et, dans l'autre branche, celles contenant v. Il parcourt ainsi toutes les possibilités de couverture (et les étapes intermédiaires pour y arriver, qui sont des couvertures partielles), calculant pour chacune sa longueur afin d'en obtenir la minimale.

Pour améliorer cet algorithme, une première possibilité, implémentée à la méthode `algoBranchementBornes`, est de ne considérer des branches où on a la possibilité de trouver une meilleure solution que celle déjà trouvée. Ainsi, à chaque branchement, on calcule une borne maximale sur la taille de la solution exacte ainsi qu'une borne minimale, et on ne branche pas (i.e., on fait un élagage) si la borne minimale pour cette branche est supérieure à la borne maximale. Les bornes maximale et minimale peuvent être calculées de plusieurs façons différentes. Notre algorithme en implémente 3 pour chaque borne.

Deux des bornes maximales utilisées consistent à calculer, à chaque étape, une couverture composée de la couverture partielle courante et d'une couverture du restant du graphe (sans les sommets de la couverture partielle) obtenue à l'aide d'un des deux algorithmes approchés. On prend ensuite le minimum entre cette valeur et la taille de la meilleure couverture déjà trouvée. L'autre borne maximale, naïve, consiste à prendre uniquement la taille de la meilleure couverture déjà trouvée.

La première borne minimale correspond à celle de l'énoncée et sera démontrée dans la Section 4.1. Cette borne utilise trois quantités, dont une qui dépend d'un calcul avec l'algorithme de couplage. Comme ce calcul peut être couteux, une deuxième borne minimale correspond à ne pas utiliser cette quantité. La troisième borne minimale, la plus naïve, correspond à prendre uniquement la taille de la couverture partielle courante.

Une autre voie d'amélioration de l'algorithme, implémentée à la méthode `algoBranchementAmeliore`, provient d'une modification du principe du branchement. En effet, en branchant sur les deux extrémités (u, v) d'une arête, la branche qui correspond à u testera toutes les couvertures comprenant u, mais la branche correspondant à v pourra brancher, plus tard, sur une arête (u, w), et aura ainsi une sous-branche qui prend également u. Donc plusieurs chemins dans l'arbre peuvent représenter en effet la même couverture. Pour éviter cela, une solution est de brancher sur un *sommet* plutôt qu'une arête : on choisit un sommet u du graphe et on fait une branche où l'on met u et une deuxième où l'on met tous les voisins de u. En effet, si u ne fait pas partie d'une couverture, alors forcément tous ses voisins en font partie, et donc cette recherche est aussi exhaustive sur tous les couvertures possibles.

Dans cet algorithme, on continue à utiliser des bornes maximales et minimales pour limiter l'exploration aux seules branches pouvant donner une solution optimale. Comme on le montrera dans la Section 4.2, la meilleure combinaison possible de bornes est avec l'algorithme approché de couplage pour la borne maximale et la borne minimale donnée à l'énoncée. Ce sont donc ces bornes que l'on utilise dans cette nouvelle version.

Dans un premier temps, on choisit u de façon arbitraire. Pour améliorer la vitesse de cet algorithme, on utilise l'heuristique de choisir u à chaque étape comme un sommet de degré maximal du graphe restant après suppression de la couverture partielle. Une autre amélioration consiste à ne pas brancher sur les sommets de degré 1 ; on montre dans la Section 4.1 pourquoi cela marche.

4.1. **Résultats théoriques.**

Soit G un graphe à n sommets et m arêtes, M un couplage de G, C une couverture de G, et Δ le degré maximum des sommets de G. Soient

$$b_1 = \left\lceil \frac{m}{\Delta} \right\rceil, \qquad b_2 = |M|, \qquad b_3 = \frac{2n - 1 - \sqrt{(2n-1)^2 - 8m}}{2}.$$

Proposition. $|C| \geq b_1$.

Démonstration. Soient $u_1, u_2, \ldots, u_{|C|}$ les sommets de C et $d_1, d_2, \ldots, d_{|C|}$ leurs degrés respectifs. Par définition de couverture, chaque arête est incidente à au moins un sommet de C, donc on a

$$\sum_{i=1}^{|C|} d_i \geq m.$$

D'autre part, on a $d_i \leq \Delta$ pour tout i, et donc

$$|C|\Delta \geq \sum_{i=1}^{|C|} d_i.$$

Combinant ces deux inégalités, on obtient que $|C|\Delta \geq m$, soit $|C| \geq \frac{m}{\Delta}$. Comme $|C|$ est un nombre entier, on conclut que $|C| \geq \left\lceil \frac{m}{\Delta} \right\rceil$. □

Proposition. $|C| \geq b_2$.

Démonstration. Comme C est une couverture de G, chaque arête de M doit avoir au moins une de ses extrémités dans C. En plus, comme M est un couplage, les extrémités des arêtes de M sont toutes différentes. Alors le nombre de sommets dans C doit être supérieur ou égal au nombre d'arêtes de M, c'est-à-dire $|C| \geq b_2$. □

Proposition. $|C| \geq b_3$.

Démonstration. Soit E l'ensemble des arêtes de G. Comme C est une couverture, alors toutes les arêtes de E doivent avoir au moins une de leurs extrémités dans C. On peut donc diviser E en deux sous-ensembles disjoints : E_1 et E_2, où E_1 est l'ensemble des arêtes dont une seule extrémité est dans C et E_2 est l'ensemble de celles dont les deux extrémités sont dans C. On peut borner la quantités d'éléments de ces deux ensembles par

$$|E_1| \leq |C|(n - |C|), \qquad |E_2| \leq \frac{|C|(|C|-1)}{2}.$$

Alors

$$\begin{aligned}
m = |E_1| + |E_2| &\leq |C|(n-|C|) + \frac{|C|(|C|-1)}{2}, \\
m &\leq |C|(n-|C|) + \frac{|C|(|C|-1)}{2}, \\
2m &\leq 2|C|(n-|C|) + |C|(|C|-1), \\
2m &\leq 2|C|n - 2|C|^2 + |C|^2 - |C|, \\
|C|^2 - (2n-1)|C| + 2m &\leq 0.
\end{aligned}$$

Le membre de gauche est un polynôme de degré 2 en $|C|$. Il est donc négatif si et seulement si $|C|$ est entre les deux racines de ce polynôme. On calcule ces racines : soit Δ le discriminant du polynôme, on a

$$\Delta = (2n-1)^2 - 8m$$

et alors les racines X_-, X_+ de ce polynôme sont

$$X_\pm = \frac{2n - 1 \pm \sqrt{(2n-1)^2 - 8m}}{2}.$$

Comme $|C|$ doit être entre ces deux racines, on a $X_- \leq |C| \leq X_+$, c'est-à-dire

$$\frac{2n - 1 - \sqrt{(2n-1)^2 - 8m}}{2} \leq |C| \leq \frac{2n - 1 + \sqrt{(2n-1)^2 - 8m}}{2}.$$

En particulier,

$$|C| \geq \frac{2n - 1 - \sqrt{(2n-1)^2 - 8m}}{2} = b_3.$$ □

À partir des trois propositions précédentes, on en déduit la borne minimale

$$|C| \geq \max\{b_1, b_2, b_3\}. \tag{1}$$

Proposition. *Dans un graphe G, si un sommet u est de degré 1, alors il existe une couverture optimale de G ne contenant pas u.*

Démonstration. Soit C une couverture optimale de G. Si $u \notin C$, on n'a rien à montrer. Si $u \in C$, comme u est de degré 1, il existe une unique arête (u, v) dans G telle que u est une de ses extrémités. Soit $C' = (C \setminus \{u\}) \cup \{v\}$. On affirme que C' est une couverture optimale de G. En effet :

- C' est une couverture de G : si on prend une arête de G, alors soit elle est égale à (u, v), et C' contient v, soit c'est une autre arête qui avait déjà une extrémité w (forcément différente de u) dans C, et cette extremité reste dans C'.
- C' est optimale : C était optimale et on a supprimé u de C et rajouté v, donc le nombre de sommets dans la couverture n'a pas augmenté, elle reste optimale. □

4.2. **Résultats expérimentaux.** Le premier algorithme implémenté a été l'algorithme de branchement de base (méthode `algoBranchement`). Afin de vérifier que cet algorithme marche correctement, on a introduit un argument facultatif dans la méthode qui l'implémente afin d'afficher ses calculs pour permettre de retracer l'arbre avec les branchements faits. On l'a ensuite testé dans l'exemple fourni avec l'énoncé qui correspond au graphe de la Figure 5, ce qui a donné le branchement de la Figure 6.

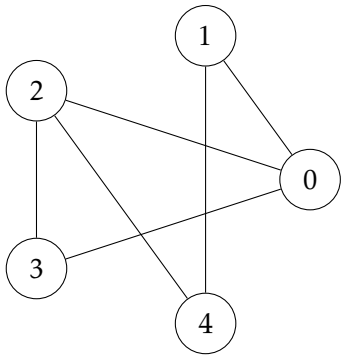


FIGURE 5. Graphe utilisé pour le test de l'algorithme de branchement

Dans la Figure 6, chaque nœud contient la couverture partielle obtenue jusqu'à présent ; on démarre ainsi de la couverture vide. À chaque branchement, on choisit une arête du graphe et on branche selon ses deux extrémités ; ainsi, à la racine, on choisit l'arête $(0, 1)$ et les deux branches correspondent aux couvertures partielles $\{0\}$ et $\{1\}$. En regardant les différences entre un nœud d'un niveau et ses fils au niveau suivant, on peut ainsi déterminer quelle arête a été choisie pour le branchement. Les nœuds à l'intérieur d'un rectangle correspondent à des feuilles : dans ces nœuds, la suppression de la couverture partielle du graphe de départ donne un graphe sans arêtes, ce qui veut dire que cette couverture partielle est une vraie couverture du graphe. Cet arbre est construit en profondeur, visitant d'abord le sous-arbre gauche et ensuite le sous-arbre droit de chaque nœud.

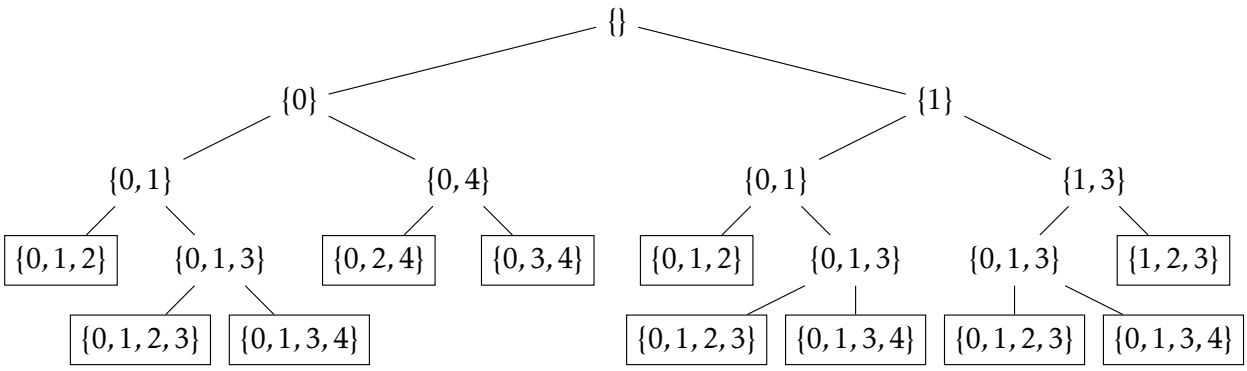
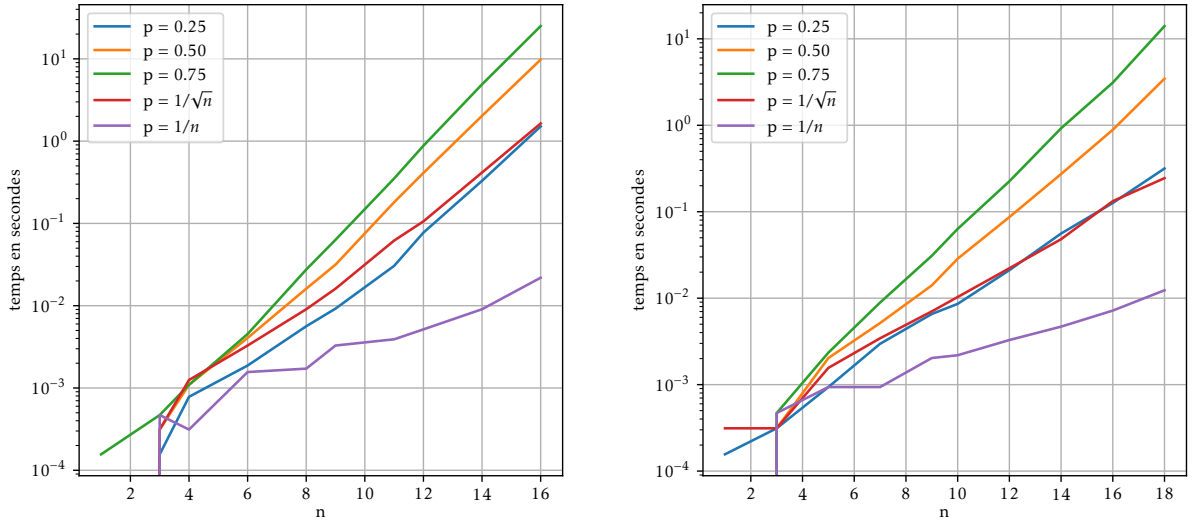


FIGURE 6. Branchement réalisé dans le graphe de la Figure 5

On remarque que l'algorithme marche comme prévu dans cet exemple : toutes les couvertures possibles du graphe apparaissent dans une feuille et on a tous les nœuds intermédiaires attendus. Cet exemple illustre aussi l'inefficacité de cet algorithme : certaines couvertures sont visitées plusieurs fois.



(a) (b)

FIGURE 7. Temps de calcul en fonction de n pour l'algorithme de branchement (a) simple et (b) avec bornes

Le temps d'exécution de cet algorithme en fonction de n et pour divers choix de p est donné dans la Figure 7(a). Avec l'échelle logarithmique dans l'axe des temps, on observe des courbes proches de droites sur la figure, ce qui suggère un comportement exponentiel en n. On n'a pu simuler que jusqu'à $n = 16$, le temps de calcul étant trop important au-delà. On remarque aussi que le temps de calcul croit avec p car une augmentation de p entraine une augmentation du nombre d'arêtes dans le graphe. On rappelle que ce nombre est en $\Theta(pn^2)$. En choisissant $p = \frac{1}{\sqrt{n}}$ ou $p = \frac{1}{n}$, on a des graphes creux, avec un nombre d'arêtes en $\Theta(n^{3/2})$ et $\Theta(n)$, respectivement.

L'utilisation des bornes minimale et maximale pour faire l'élagage lors du parcours de l'arbre de branchement a été implémentée dans la méthode `algoBranchementBorne` et les résultats du temps de simulation avec cet algorithme sont données dans la Figure 7(b). Cette figure utilise comme borne maximale celle basée sur l'algorithme de couplage et, comme borne minimale, (1). On remarque, en comparant les Figures 7(a) et (b), que l'introduction des bornes donne un gain de temps : on a pu aller jusqu'à $n = 18$ avec le même ordre de grandeur de temps qu'on avait pour $n = 16$ sans l'élagage, qui est de l'ordre de $10s$ pour le cas $p = 0.75$. Cela n'améliore cependant pas la complexité, les courbes obtenues suggérant toujours un comportement exponentiel.

On a aussi implémenté les autres bornes maximales et minimales décrites au début de la Section 4, les résultats de temps moyen de simulation correspondants pour $p = 0.25$ étant présentés dans la Figure 8. Dans la légende de cette figure, on représente les bornes maximale et minimale utilisées dans la simulation par un couple (i, j), où i correspond à la borne maximale et j à la minimale. Le cas $i = 0$ correspond à la borne maximale obtenue à l'aide de l'algorithme de couplage, $i = 1$ à celle avec l'algorithme glouton, et $i = 2$ à la borne naïve. Le cas $j = 0$ correspond à la borne minimale (1), $j = 1$ à (1) sans le terme b_2 (qui nécessite un calcul de l'algorithme de couplage), et $j = 2$ à la borne naïve.

On observe que, avec la borne minimale (1) ($j = 0$) fixée, le temps de simulation est minimal avec la borne maximale donnée par l'algorithme de couplage, et les temps pour la borne naïve ou par l'algorithme glouton sont comparables. En effet, l'algorithme glouton, comme vu à la Section 3.2, est typiquement plus proche de l'optimal que l'algorithme de couplage, mais son temps d'exécution est plus important, donc le gain d'élagage qu'on peut avoir n'est pas suffisant pour

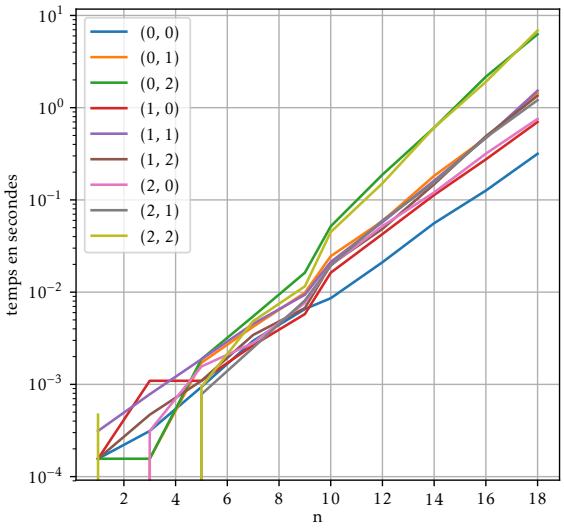
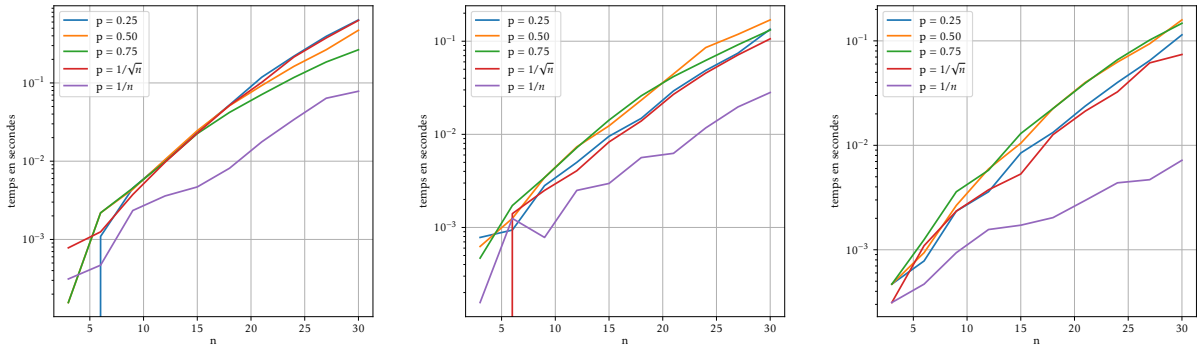


FIGURE 8. Temps de calcul en fonction de n pour des variantes de l'algorithme de branchement avec bornes

compenser son temps d'exécution plus long et, à la fin, il est pire que l'algorithme de couplage. La borne naïve est plus rapide à calculer, mais permet de faire moins d'élagage, et à la fin elle fait aussi moins bien que la borne par couplage. Les autres choix de bornes minimales rendent la situation pire : tous les cas avec (1) sans b_2 ($j = 1$) s'exécutent dans essentiellement le même temps, indépendamment de la borne supérieure, et ce temps est supérieur à tous les cas avec $j = 0$. Considérer la borne minimale naïve ne change pas le temps d'exécution lorsque la borne supérieure est l'algorithme glouton, mais l'augmente dans tous les autres cas de borne supérieure. Ainsi, le cas $(0, 0)$ correspond à la meilleure situation.



(a) (b) (c)

FIGURE 9. Temps de calcul en fonction de n pour l'algorithme de branchement amélioré (a) simple, (b) avec choix d'un sommet de plus grand degré en premier, et (c) avec élimination des branches avec un sommet de degré 1.

Les améliorations de l'algorithme de branchement pour éviter de parcourir deux fois la même couverture ont été implémentées dans la méthode `algoBranchementAmeliore` et les résultats correspondants de temps de simulation moyens sont donnés dans la Figure 9. On observe que ces améliorations rendent l'algorithme beaucoup plus rapide : on peut aller jusqu'à $n = 30$ avec des temps de simulation inférieurs à $1s$. Le temps de simulation semble dépendre très peu de p : dans les graphes avec beaucoup d'arêtes, le fait de brancher entre un sommet et tous ses voisins permet de prendre, dans une des branches, un grand nombre de sommets en même temps, ce qui diminue la taille de l'arbre de branchement. Seul le cas $p = 1/n$ donne un temps de simulation beaucoup

plus rapide, car le graphe correspondant est assez creux et l'arbre de branchement sera donc assez petit.

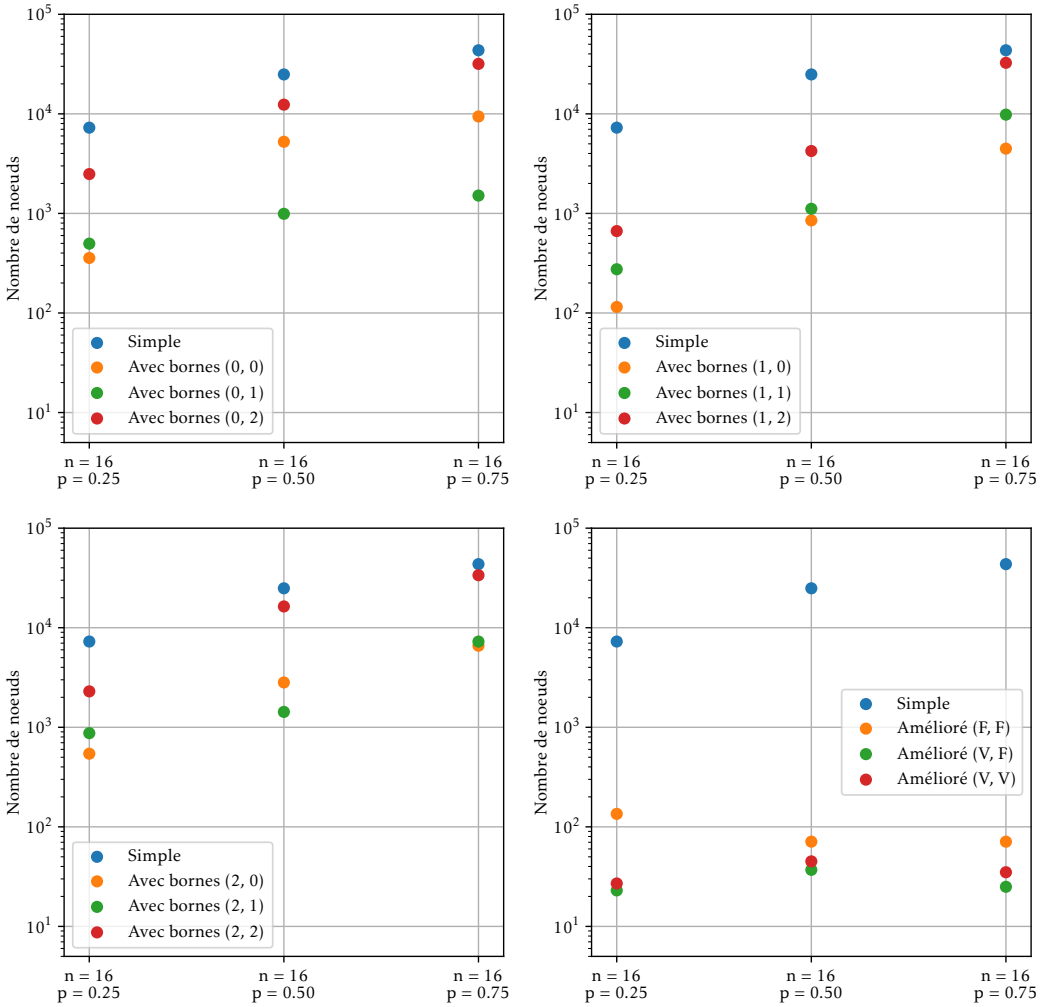


FIGURE 10. Nombre de nœuds visités pour les différents algorithmes

On a également comparé l'ensemble des algorithmes de branchement par rapport au nombre de nœuds visités dans l'arbre de branchement pour des graphes aléatoires avec $n = 16$ et probabilité d'existence d'une arête $p \in \{0.25, 0.50, 0.75\}$. Les résultats correspondants sont données dans la Figure 10. Dans la légende, pour l'algorithme amélioré, les cas (F, F), (V, F) et (V, V) correspondent, respectivement, à l'algorithme amélioré de base, à la version en prenant les sommets pour le branchement par ordre décroissant de degré, et à la version précédente avec en plus la suppression des branches de degré 1.

On remarque que l'algorithme simple visite beaucoup de nœuds, même dans ce cas avec 16 sommets, le nombre de nœuds visités étant de l'ordre de 10^4 à 10^5. Les algorithmes avec élagage par bornes minimale et maximale font souvent mieux, mais ne permettent pas d'avoir moins que 10^2 nœuds. Ce sont les algorithmes améliorés qui permettent les gains les plus importants (comme déjà attendu par leur temps d'exécution beaucou plus rapide) : on peut gagner entre 2 et 3 ordres de grandeur, avec une quantité de nœuds visités entre 10 et 100 sur ces instances.