Couverture d'un graphe non orienté (Vertex Cover)

Virginie Chen, Paul-Tiberiu Iordache

21 Octobre 2023

Introduction

Dans le cadre de l'unité d'enseignement COMPLEX (Complexité, algorithmes randomisés et approchés - MU4IN900) de *Sorbonne Université*, nous avons étudié le problème Vertex Cover. Le but de ce problème NP difficile est de trouver une couverture d'un graphe non orienté G = (V,E) contenant un nombre minimum de sommets. Notre choix de langage de programmation a été Python pour l'utilité de la librarie numpy dans ce projet. Nous sommes conscients que le temps de calcul peut être plus long par rapport à d'autres langages comme le C ou le C++. Cependant, étant donné que le but de ce projet n'est pas d'avoir un temps d'éxécution le plus rapide, mais plutôt de comparer le temps d'éxécution des différents algorithmes testés sur différentes instances plus ou moins grande, nous ne pensons pas que choisir Python soit pénalisant. En effet, si un algorithme A est 3 fois plus lent, l'algorithme B sera également 3 fois plus lent dû à Python.

Contents

1 Contextualisation

1.1 Définition d'un graphe non orienté G = (V,E)

Soit $G = (V, E)$ un graphe non orienté.
V est l'ensemble des n sommets du graphe et E l'ensemble de ses m arêtes.

1.2 Définition d'une couverture

Une couverture de G est un ensemble $V_0 \subseteq V$ tel que toute arête $e \in E$ a (au moins) une de ses extrémités dans V.

2 Graphes

2.1 Choix d'implémentation de la structure du graphe G = (V,E) non orienté

Pour l'implémentation de la structure d'un graphe non orienté, nous avons eu deux idées différentes.

La première est de créer une classe Graph où nous stockons les sommets dans un ensemble **V** et l'ensemble d'arêtes dans un dictionnaire **E** dont la clé est l'identifiant du sommet, et les valeurs sont un ensemble de sommets adjacents. Par exemple, si l'arête (i, j) existe, alors dans notre dictionnaire, cela ressemblerait à :

{ i : {j}, j : {i} }

La deuxième idée était d'utiliser une matrice binaire des sommets où chaque case de cette matrice est définie par $M[i][j] = 1$ et $M[j][i] = 1$ si l'arête entre les sommets i et j existe.

Finalement, nous avons choisi la première option car l'utilisation de la programmation orientée objet en python pour implémenter un graphe nous semblait élégante et plus adaptée à la question de la suppression de sommets. Par exemple, si nous avons un graphe avec 4 sommets $V = \{0, 1, 2, 3\}$ et que nous devons supprimer le sommet 2, avec l'implémentation basée sur des ensembles, on aurait $V = \{0, 1, 3\}$. L'implémentation avec une matrice ne nous permettrait pas cela. De plus, la vérification de l'appartenance d'un élément à un ensemble s'effectue en O(1). Et enfin, il n'est pas possible de faire des doublons de sommets avec les ensembles et la structure d'un graphe de cette manière nous semblait, personnellement plus clair.

2.2 Opérations de base

Pour la question 2.1.3, il était demandé de coder une méthode renvoyant les degrés des sommets des graphes sous forme de tableau. Etant donné nos choix d'implémentation, il était plus simple et également plus clair pour nous de renvoyer un dictionnaire. En effet, au lieu de renvoyer un tableau, nous renvoyons un dictionnaire de la forme **{sommet : degré}**. De ce fait, on peut connaître le sommet associé au degré en question, ce qui est, pour nous, une meilleure décision d'implémentation dans le cadre des choix qui ont été faits.

3 Méthodes approchées

3.1 Algorithme Glouton

Le principe d'un algorithme glouton est de faire des choix optimaux à l'instant t, sans prendre en compte les conséquences futures. Généralement, à première vue, un algorithme glouton nous donne des résultats assez satisfaisants sur la question de l'optimalité. Cependant, il ne vérifie pas cette condition d'optimalité dans tous les cas.

Nous avons trouvé un contre exemple pour lequel l'algorithme glouton ne nous renvoie pas la solution optimale. Dans la Figure 1 nous observons un graphe non orienté comportant 9 sommets. Il est facile de voir qu'une des couvertures optimale V_0 possible est :

$$V_0 = \{1, 3, 6, 7\}$$

Or, l'algorithme glouton que nous avons codé nous renvoie l'ensemble suivant :

$$C = \{0, 1, 3, 6, 7\}$$

Autrement dit, l'algorithme glouton renvoie un sommet en plus que la couverture minimale. Il n'est donc pas optimal.

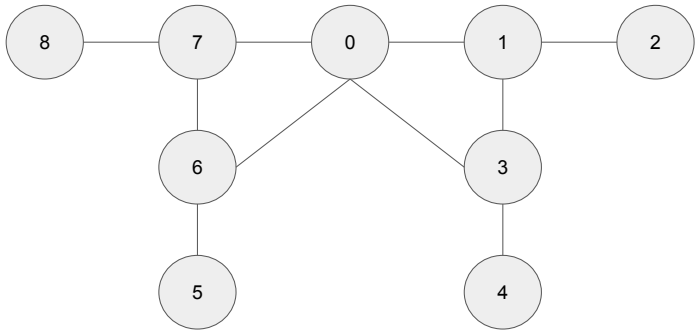


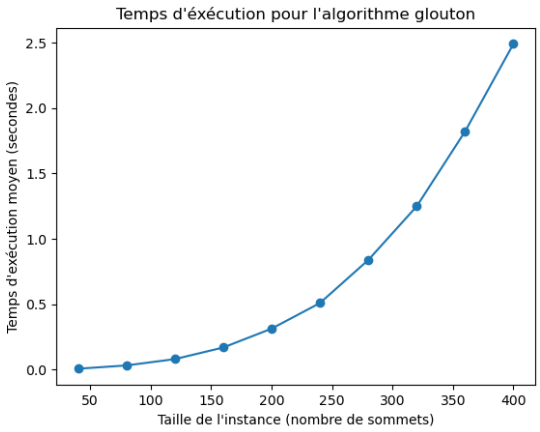
Figure 1: Graphe non orienté à 9 sommets où glouton n'est pas optimal

Nous avons vu que dans cet exemple que la couveture minimale comporte 4 sommets.
Glouton nous renvoyant 5 sommets, on peut dire que glouton est au moins $\frac{5}{4}$ - approché. Autrement dit, glouton renvoie au moins $\frac{5}{4}$ fois plus de sommets que dans l'optimale. Choisissons arbitrairement $\frac{9}{8} \leq \frac{5}{4}$. Dans cet exemple, glouton se trompe de plus que $\frac{9}{8}$. Il ne peut donc pas être $\frac{9}{8}$ - approché, sinon il ne pourrait pas faire une erreur plus grande que $\frac{9}{8}$ fois plus de sommets que la couverture optimale.

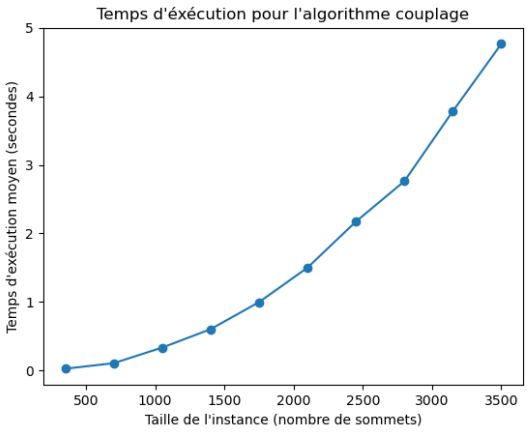
3.2 Comparaison de glouton et couplage

Il est très rapide de constater que l'algorithme couplage est bien plus rapide au niveau du temps d'éxécution que glouton. Nous allons formaliser cette constatation.

Tout d'abord, nous avons généré des graphes aléatoires ayant n sommets qui lient deux sommets entre eux avec une probabilité p donnée. Ensuite, il a fallu pour mesurer le temps d'éxecution, nous fixer une certaine limite de sommets N_{max}. Celle-ci a été choisie de sorte que l'algorithme puisse s'éxécuter en quelques secondes. Nous trouvons donc $N_{Gmax} = 400$ sommets pour glouton qui s'éxécute alors en ≈ 2.4216 secondes. Et $N_{Cmax} = 3500$ sommets pour couplage qui s'éxécute alors en ≈ 2.7556 secondes. [1] Enfin, nous avons tracé des courbes. Cependant, étant donné que l'on se base sur des graphes générés aléatoirement, nos points étaient complètement dispersés. Il a donc fallu faire une moyenne sur une dizaine d'instances pour chaque taille d'instance pour la mesure du temps d'éxécution. Ce qui nous donne après tout, une courbe qui est claire. Nous avons choisi arbitrairement une probabilité de 0.3 pour finalement obtenir les courbes suivantes :



(a) Algorithme glouton



(b) Algorithme couplage

Figure 2: Courbes du temps d'éxécution moyen en fonction de la taille de l'instance

Comme évoqué dans le 3.1, notre algorithme glouton devient relativement lent lorsqu'on lui donne des tailles d'instances un peu plus grandes. Son N_{Gmax} est bien plus petit que N_{Cmax} et cela se constate sur nos courbes où glouton devient très rapidement lent. Il atteint une seconde d'éxécution pour environ 300 sommets, tandis que pour couplage on peut aller jusqu'à presque 2000 sommets pour rester à un temps d'éxécution d'une seconde.

Avec nos deux graphiques, il n'est pas possible tout de suite de déterminer si le temps d'éxécution pour chaque algorithme est exponentiel ou plutôt polynomial. Pour pouvoir identifier exactement nos courbes, il faut passer en échelle logarithmique.
En effet, si on utilise les propriétés du logarithme, on a :

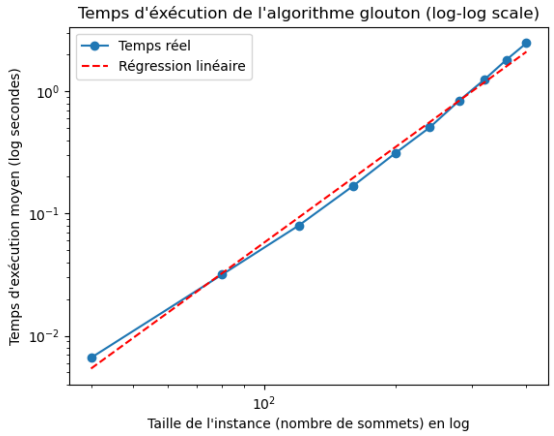
$$\begin{cases} \log(y) = \log(b^x) = x\log(b), & \text{dans le cas exponentiel, où b est la base} \\ \log(y) = \log(x^n) = n\log(x), & \text{dans le cas polynomial, où n est le degré} \end{cases}$$

[1] Il est important de noter que ces temps d'éxécution varient d'une fois à l'autre selon les graphes sur lesquels on applique l'algorithme.

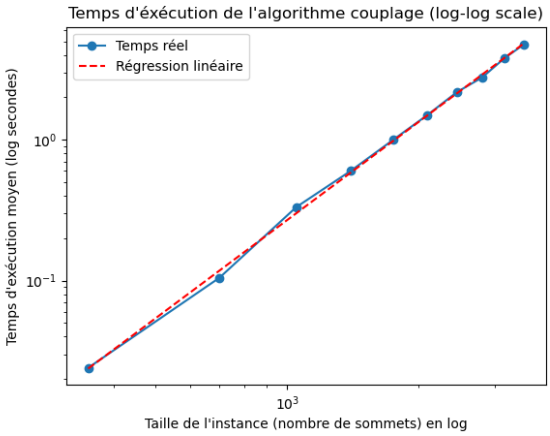
De ce fait,

- Si on se trouve avec une courbe **polynômiale**, alors lorsqu'on exprime la relation $\log(y) = \log(x)$ on doit obtenir une droite dont la pente nous donne le degré n du polynôme.
- Si on se trouve avec une courbe **exponentielle**, alors lorsqu'on exprime la relation $\log(y) = x$ on doit obtenir une droite dont la pente nous donne $\log(b)$, donc $e^{\log b}$ est la base de notre exponentielle.

En l'occurence, nous avons essayé les deux cas pour déterminer si nos courbes étaient plutôt polynômiales ou alors, exponentielles. Lorsque nous avons essayé d'exprimer la relation $\log(y) = x$, la courbe, que ce soit pour glouton ou pour couplage, devient de type logarithmique. Toutefois, lorsque nous avons appliqué le logarithme sur les deux axes, x et y, nous avons obtenu une droite pour chacun de nos deux algorithmes, comme illustré sur la Figure 4 . Reste donc à déterminer la pente de nos deux droites pour en déduire le degré des polynômes respecitfs.



(a) Algorithme glouton



(b) Algorithme couplage

Figure 3: Courbes du temps d'éxécution moyen en échelle logarithmique sur x et y

Pour déterminer les degrés de nos deux polynômes, nous avons d'abord effectués une régression linéaire pour avoir une pente correctement ajustée aux points obtenus. Puis nous avons utilisés une fonction de la bibliothèque Numpy pour calculer notre coefficient directeur, ou autrement appelé, pente. Nous trouvons ainsi comme résultats :

```
Pente de la régression linéaire: 2.59
```

(a) Degré de glouton

```
Pente de la régression linéaire: 2.31
```

(b) Degré de couplage

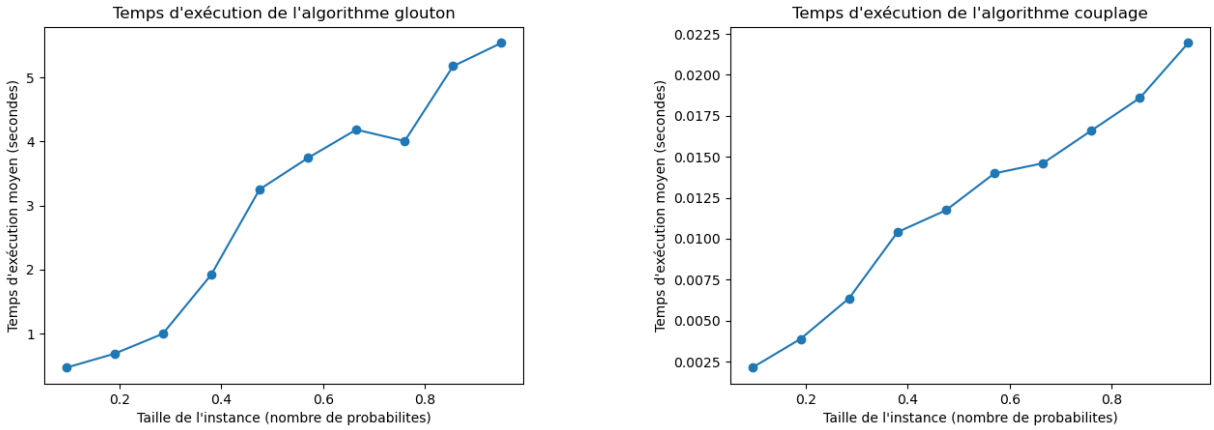
Figure 4: Les degrés respectifs des polynômes obtenus par glouton et couplage

On peut observer que les degrés sont relativement proches. Ils sont tout de même cohérents avec nos graphiques originels de la Figure 2. En effet, sur ces premières courbes, on avait reconnu que le temps d'éxécution de l'algorithme glouton augmentait bien plus rapidement selon la taille de l'instance par rapport à l'algorithme couplage. Donc l'algorithme glouton devrait avoir un degré plus important. Ce qui est effectivement le cas $2.59 \geq 2.31$. On aurait pu toutefois s'attendre à un écart bien plus important entre les degrés des deux polynômes étant donné cette différence de

croissance. Néanmoins, cela reste cohérent car le petit écart pourrait s'expliquer par des constantes cachées.

Nous avons également comparé la qualité des résultats obtenus avec glouton et avec couplage. Pour cela, nous avons utilisé une boucle qui tourne tant que la taille de la solution retournée par glouton reste plus petite que celle retournée par couplage. Ainsi, on observe que glouton renvoie toujours une couverture avec un nombre de sommets inférieur ou égal à couplage. Nous sommes allés jusqu'à 407 sommets[2] pour la boucle. On peut donc dire que dans la plupart des cas, glouton renvoie une solution plus satisfaisante que couplage.

Nous venons d'analyser l'évolution du temps d'éxécution en fonction de la taille des instances pour une probabilité p donnée. Cependant, il est également possible d'analyser l'évolution du temps d'éxécution en faisant varier les probabilités pour un nombre[3] de sommets donné. En effectuant exactement la même démarche que lorsque nous avons fait varier les sommets, nous obtenons les courbes de la Figure 5.



(a) Algorithme glouton

(b) Algorithme couplage

Figure 5: Courbes du temps d'éxécution moyen lorsqu'on fait varier p

La courbe obtenue lorsqu'on fait varier les probabilités pour l'algorithme glouton est très aléatoire. Ici, on observe un des cas possibles. Cette irrégularité s'explique par une incertitude statistique. En effet, nous faisons la moyenne du temps d'éxécution sur dix instances différentes pour chaque probabilité. Toutefois, il semblerait que ce ne soit pas suffisant[4]. On suppose que c'est pour cette raison qu'on observe en 5a, une courbe distordue.

Puisque nous avons choisi le nombre de sommets en fonction de glouton, ce dernier étant celui qui nous limite en terme de temps d'attente, couplage, lui, est beaucoup plus rapide. Il a donc été possible d'augmenter largement le nombre d'instances[5] sur lequel on fait une moyenne. Par conséquent, il est inévitable de constater que la courbe 5b présente bien moins d'irrégularité statistique.

Il est tout de même possible de remarquer qu'il semble y avoir une évolution du temps d'éxécution

[2]Au délà, le temps de calcul devenait trop long

[3]Nous avons choisi 200 sommets par rapport à glouton qui devient rapidement lent.

[4]Pour autant, dû à notre fonction génératrice de graphe, il n'est pas possible de choisir une moyenne avec plus d'instance car ce serait beaucoup trop long.

[5]La moyenne a été faite sur 100 instances.

relativement linéaire en fonction des probabilités. Cela s'explique par le fait que lorsqu'on choisi une probabilité plus grande, le graphe généré contient plus d'arêtes que lorsqu'on a une probabilité plus faible. Ainsi, nous avons plus d'arêtes à couvrir et les algorithmes sont linéairement plus lents.

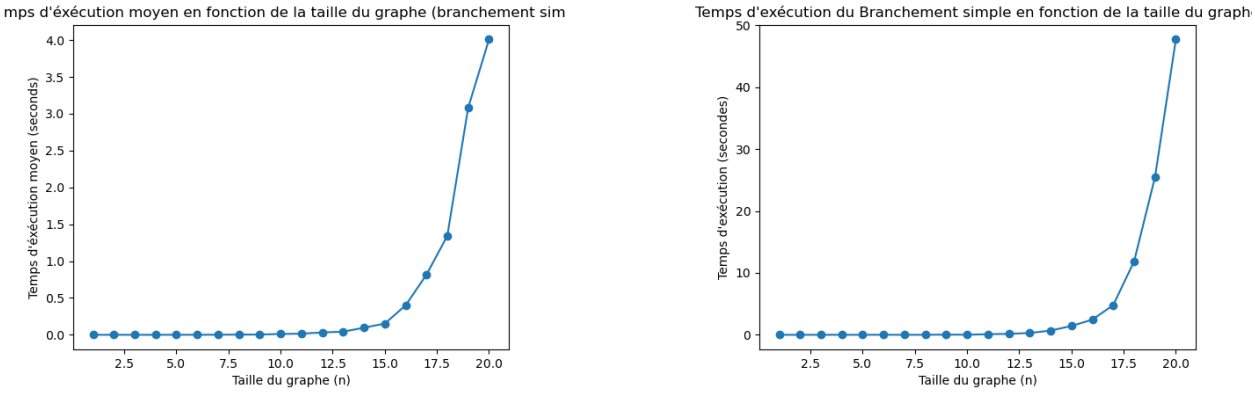
4 Branchement

4.1 Branchement simple

Suite au codage de notre méthode de branchement simple, nous avons effectué des tests sur celle-ci en faisant varier le nombre de sommets jusqu'à $N_{max} = 20$. Nous avons reproduit exactement la même démarche que dans la section 3.2. Les deux probabilités choisies pour tester la méthode sont $P_1 = \frac{1}{\sqrt{N_{max}}}$ et $P_2 = 0,7$. Les courbes obtenues sont représentées dans la Figure 6. Mais tout d'abord, expliquons ces choix.

Il était prévu au départ de comparer les courbes obtenues à probabilité $P1$, $P2$ et 0,3. Nous avons finalement décidé de garder uniquement les deux extrêmes car ceux-ci représentaient parfaitement l'influence de la probabilité sur le temps d'éxécution. Pour la probabilité 0,3 on observait uniquement un juste milieu du comportement du temps d'éxécution entre P_1 et P_2 qui a donc été jugé, non nécessaire, pour notre analyse.
Quant au choix du Nmax, il a été fait en considérant le temps d'éxécution pour la courbe 6b. En effet, comme on le voit très clairement, le temps d'éxécution nécéssaire pour 20 sommets est déjà de 50 secondes. Ainsi, si on avait choisi un nombre de sommets supérieur, il aurait été trop long d'attendre la fin du programme. C'est également pour cette raison qu'il n'a pas été choisi de mesurer le temps d'éxécution moyen, encore une fois, le temps d'attente aurait été trop long. De plus, la courbe est assez satisfaisante telle quelle. En revanche, ce n'est pas le cas pour la probabilité $\frac{1}{\sqrt{N_{max}}}$.

Pour P_1, il était nécessaire de calculer un temps d'éxécution moyen sur 15 instances car tout comme dans le cas de la Figure 5, nous retrouvions des anomalies pour certains sommets. Ce sont encore une fois, des erreurs statiques qui s'atténuent en calculant des moyennes. Le temps d'éxécution étant largement moindre, cela était réalisable.



(a) Temps d'éxécution moyen pour P_1

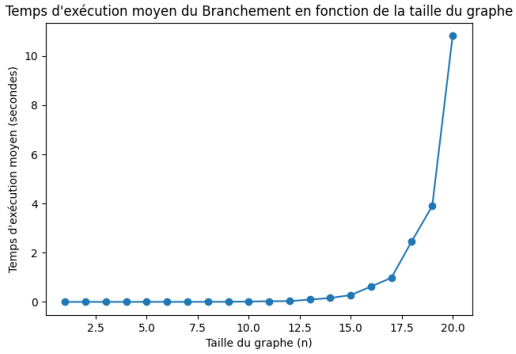
(b) Temps d'éxécution pour P_2

Figure 6: Courbes du temps d'éxécution moyen lorsqu'on fait varier P

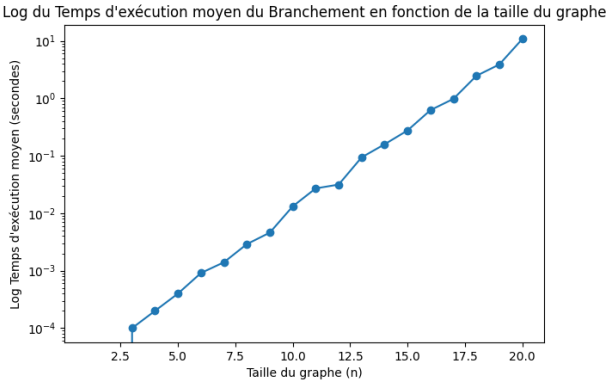
Les courbes ont toutes les deux un comportement très similaire. On observe un sommet déclancheur pour lequel la courbe monte très vite. La différence réside sur le temps qu'elles atteignent finalement. En effet, pour la courbe 6a, nous atteignons les 4 secondes pour 20 sommets tandis que pour la courbe 6b, nous montons à 50 secondes. Lorsqu'on a une probabilité plus élevée, malgré la même allure, la courbe monte plus haut. L'explication est la même que pour la comparaison couplage/glouton, figure 5. Bien que la probabilité influe sur le temps d'éxécution, elle ne paraît pas pertinente car on voit que le changement drastique se fait pour un sommet déclancheur commun, 15. Même en faisant varier les probabilités, il est clair que c'est le nombre de sommets choisis pour l'instance qui augmente significativement le temps de calcul.

4.1.1 Analyse de la courbe

Tout comme dans la section 3.2, nous allons pousser notre analyse pour déterminer si nos courbes qui ont l'air exponentielles le sont vraiment. On choisira $p = 0.3$ et 10 instances pour la moyenne. On applique donc exactement la même méthode. On obtient les courbes de la Figure 7.

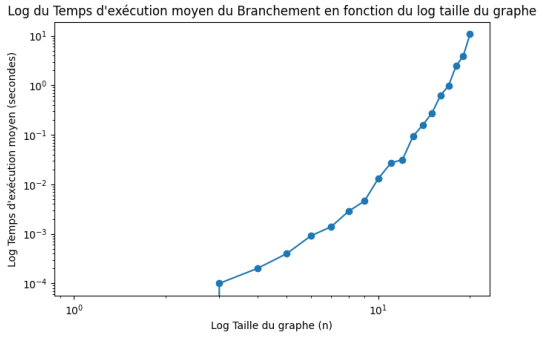


(a) Temps d'éxécution moyen y en fonction de x



(b) Temps d'éxécution moyen log scale sur y

Figure 7: Courbes obtenus avec l'algorithme branchement simple



Lorsque nous avons tracé une échelle logarithmique sur x et y, on a obtenu une courbe qui est restée de type exponentielle. A l'inverse, lorsque nous avons tracé en échelle logarithmique sur y, nous avons obtenu une droite. Ce cas de figure correspond bien à une courbe exponentielle sur le graphe d'origine. Nous venons donc de confirmer que le temps d'éxécution selon la taille de l'instance de branchement simple correspond à une évolution exponentielle. Il faut maintenant en déterminer sa base.

Pour le calcul de la pente, on utilise cette fois la fonction linegress de la bibliothèque scipy.stats. Comme expliqué précédemment dans la section 3.2, on obtient non pas la base de notre exponentielle mais le logarithme de celle ci $\log(b)$. Ainsi notre **coefficient directeur** vaut ≈ 0.66.

Cette valeur nous atteste encore une fois que l'on se trouve bien dans le cas d'une fonction exponentielle. Puisque c'est un logarithme, on s'attendait à une valeur très faible. Ce qui s'avère être le cas.
On applique finalement l'exponentielle pour trouver la **base** de notre courbe initiale : $b \approx 1.94$.

4.2 Ajout de bornes

Notre première ébauche de branchement renvoie une solution optimale comme souhaitée. En revanche, elle devient trop rapidement lente. Il n'est pas possible d'étudier de grandes instances. C'est pourquoi, nous allons tenter de l'améliorer. Pour cela, nous introduisons des bornes. Une borne supérieure qui sera égale à la taille de la solution renvoyée par l'algorithme couplage, et une borne inférieure correspondant à $max\{b_1, b_2, b_3\}$. Avant toute chose, il est nécessaire de vérifier la validité de chacunes des bornes b_1, b_2 et b_3.

* b_1 : On considère Δ le degré maximum des sommets du graphe. Autrement dit, chaque sommet peut couvrir **au plus** Δ arêtes. C étant une couverture, par définition, toutes les arêtes du graphe sont couvertes par les sommets de C. Si on considère le cas le plus extrême où tous les sommets dans C ont Δ arêtes à couvrir, alors $|C|\Delta \geq m$. Donc $|C| \geq \frac{m}{\Delta}$. Enfin, $|C|$ étant un nombre entier, on a :
$$\lceil \frac{m}{\Delta} \rceil \leq |C|$$
$b_1 = \lceil \frac{m}{\Delta} \rceil$ est done une borne valide.

* b_2 : Soit M un couplage de taille $|M|$. Chaque arête de M n'a pas d'extrémité en commun avec une autre arête de M. Ainsi pour chaque arête e_i pour $i \in [\![1, |M|]\!], \exists s_j \in C$ pour $j \in [\![1, |C|]\!]$ tel que s_j couvre e_i, et uniquement e_i. En effet, si s_j couvre une arête e_{i_2} pour $i_2 \neq i$, cela voudrait dire que e_{i_2} a pour extrémité s_j. Or s_j est déjà extrémité de e_i, donc s_j ne couvre que e_i, sinon M n'est pas un couplage. Finalement, on peut en déduire :
$$|M| \leq |C|$$

* b_3 : Considérons la couverture C contenant $|C| = c$ sommets. On va d'abord se poser la question de combien d'arêtes on peut couvrir maximum avec ces c sommets. Le nombre maximum d'arêtes où C est une couverture correspond au résultat de la somme $\sum_{k=1}^{c}(n-k)$. On remarque que c'est une suite arithmétique qui somme c termes. Le résultat est donc $m_{max} = \frac{c}{2}(2n-1-c)$. Par conséquent, on a l'inégalité : $m_{max} \geq m$. En développant et simplifiant notre expression on a : $-c^2 + (2n-1)c - 2m \geq 0$. Ce qui nous ramène donc à déterminer les bornes pour lesquels le polynôme défini est positif.
$\Delta = (2n-1)^2 - 8m$ [6]. Les racines sont $C_{min} = \frac{-(2n-1)+\sqrt{(2n-1)^2-8m}}{-2}$ et $C_{max} = \frac{-(2n-1)-\sqrt{(2n-1)^2-8m}}{-2}$.
a, le coefficient de c^2 est égal à -1. Suite aux propriétés sur les polynômes on trouve l'inégalité
$$C_{min} \leq c \leq C_{max}$$
En gardant la partie gauche de notre encadrement on trouve :
$$\frac{2n-1+\sqrt{(2n-1)^2-8m}}{2} \leq |C|$$
Ce qui conclut notre preuve.

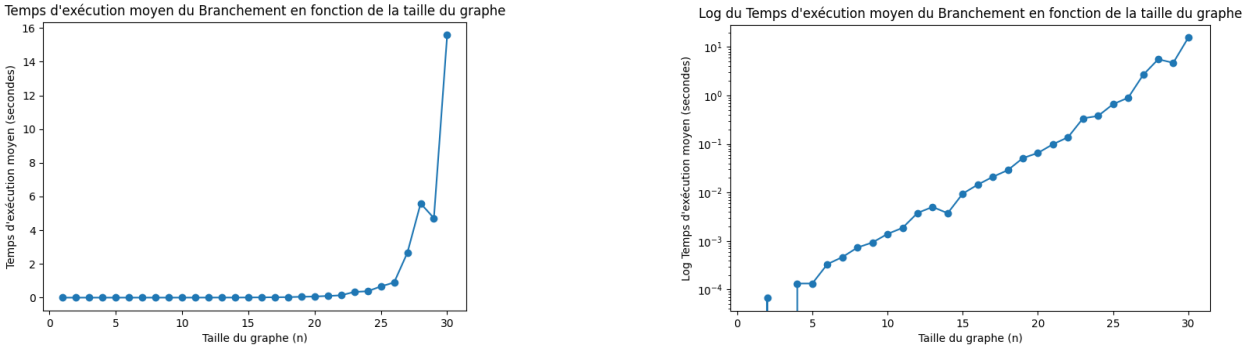
En conclusion, $B_{inf} = max\{b_1, b_2, b_3\}$.

[6] $(2n-1)^2 - 8m \geq 0$ car au plus $8m = \frac{8n(n-1)}{2} = 4n^2 - 4n$ et $(2n-1)^2 = 4n^2 - 2n + 1$

4.2.1 Branch and bound

Pour vérifier que notre ajout d'une B_{sup} et d'une B_{inf} a effectivement rendu notre algorithme plus rapide, nous allons procéder à la même analyse que pour branchement simple. Ce processus va être itéré pour chaque amélioration du branch and bound.

Tout comme pour branchement simple, la courbe en échelle log log reste courbée. Il est vrai que nos courbes de la figure 8 présentent quelques anomalies statistiques. Cependant, ayant choisi de calculer le temps moyen d'éxécution sur 15 instances et 30 sommets[7], atténuer ces erreurs plus que ce que l'on observe serait très long, c'est pour cette raison que nous avons fait le choix de les laisser telles quelles.



(a) Temps d'éxécution moyen y en fonction de x

(b) Temps d'éxécution moyen log scale sur y

Figure 8: Courbes obtenus avec l'algorithme branch and bound

Comme le montre la courbe 8b on a une droite. Lors du calcul de la pente de celle-ci, nous avons rencontrés quelques problèmes. Le branch and bound amélioré semblant plus efficace, pour les sommets proches de 0, on obtient des temps d'éxécution extrêmement rapides. Ces derniers apparaissent alors dans notre tableau de valeurs comme un temps égal à 0. Cependant, lorsqu'on applique l'échelle logarithmique cela pose une difficulté notoire. C'est pourquoi, nous allons ajouté une partie dans notre code pour pouvoir récupérer uniquement les temps d'éxécution assez grands[8]. Evidemment, on supprime ensuite les tailles du graphe associés au temps d'éxécution supprimés. Suite à cette manipulation, on procède de la même manière que pour branchement simple pour avoir la base. Au bout du compte, $b \approx 1.53$.

La base plus petite nous témoigne que le branch and bound est plus rapide et cela peut se voir également en comparant les figures 8a et 7a.

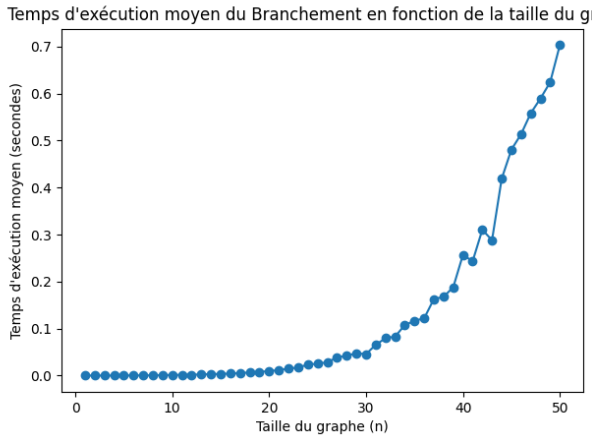
4.3 Amélioration du branchement

Dans cette section, nous améliorons notre dernière version du branch and bound en deux étapes. Tout d'abord, lorsqu'on considère une arête (u, v), quand on prend u dans la première branche, on suppose que dans la deuxième où on prend v, on ne prend pas u (on prend alors tous les voisins de u). Ensuite, nous améliorons encore cette version en choisissant une arête (u, v) tel que u soit le sommet au degré maximum. Pour ces deux algorithmes on obtient les courbes de la figure 9. Il est

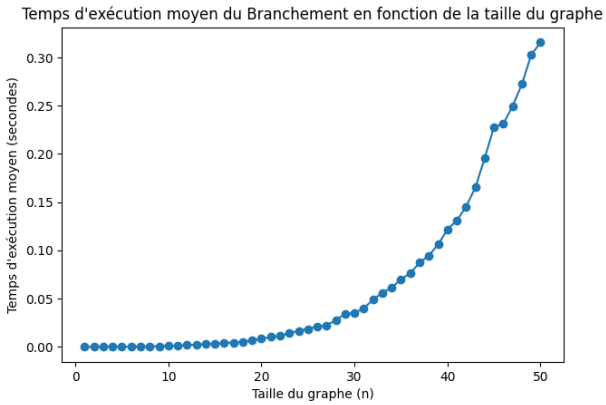
[7] la probabilité est toujours à 0.3.

[8] affichés non nuls.

évident que l'algorithme privilégiant le degré maximal s'avère plus rapide que l'approche améliorée de branch and bound, car sa durée moyenne d'exécution est plus courte que celle de l'algorithme amélioré



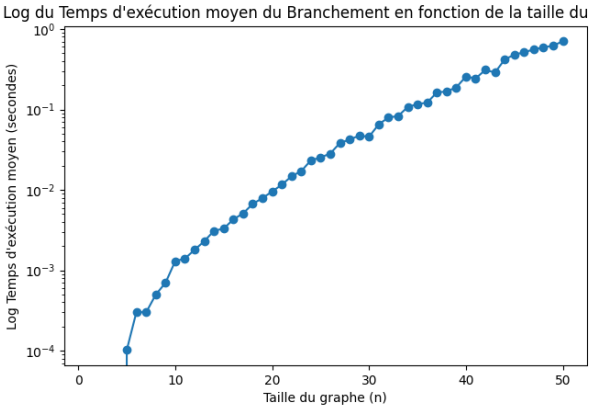
(a) Courbe pour branch and bound amélioré



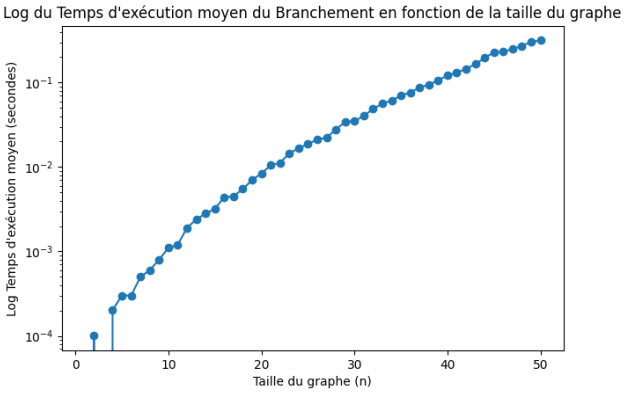
(b) Courbe pour branch and bound amélioré degré max

Figure 9: Courbes des deux algorithmes branch and bound améliorés

On s'attend donc, pour notre dernière amélioration de l'algorithme à une base de l'exponentielle qui diminue. Pour comparer ces deux bases, nous allons coder une méthode qui est sensiblement la même que celle qu'on a utilisé pour obtenir nos courbes, tout en ayant la différence que pour chacun des algorithmes branch and bound amélioré nous utiliserons exactement les même graphes à chaque fois pour calculer le temps d'éxécution moyen. On trouvera les droites obtenus avec cette nouvelle méthode dans la figure 10 ci-dessous.



(a) Droite pour branch and bound amélioré



(b) Droite pour branch and bound amélioré degré max

Figure 10: Droites des deux algorithmes branch and bound améliorés en utilisant les mêmes graphes

Au final, on trouve les bases :

- Pour **branch and bound amélioré** $b \approx 1.1924$
- Pour **branch and bound degré max** $b \approx 1.1627$

Dès lors, il est clair que le branch and bound degré max présente une base exponentielle plus faible que le branch and bound amélioré.

4.4 Qualité des algorithmes approchés

Nous avons codé une méthode permettant de calculer le rapport entre la solution d'un des algorithmes[9] et la solution optimale renvoyée par un de nos branch and bound. On cherche expérimentalement, le pire rapport d'approximation. Pour ceci, dans notre méthode, nous prenons en paramètre un nombre de sommet n, un paramètre p, et la version du branch and bound que l'on veut utiliser. Il est évident que nous avons choisi pour nos tests de prendre la version finale du branch and bound. Bien que tous nos algorithmes nous renvoient une solution optimale, ils ne prennent pas le même temps pour le faire. Etant donné que pour avoir le pire rapport d'approximation possible, il est plus intéressant de faire les tests le plus de fois possible, il est important que le temps de calcul soit rapide.

Lors des tests nous avons choisi la probabilité 0.3 pour ne pas avoir trop d'arêtes à couvrir. Cependant, il arrive que pour un nombre de sommets très faible[10], notre méthode branch and bound nous retourne une solution vide. Le graphe généré n'ayant pas assez d'arêtes. Cela se traduit ensuite par une division par 0. Ainsi pour éviter au maximum ce problème, il a été choisi pour notre boucle de approx_ratio d'itérer de 3 à n. On trouve dans le tableau 1 le pire rapport d'approximation pour glouton et couplage, selon des tailles de n différentes.

N	Pire rapport d'approximation	
	algo_glouton	algo_couplage
10	1.0	≈ 1.66
30	≈ 1.04	1.25
50	≈ 1.02	≈ 1.19

Table 1

Finalement :

- Le pire rapport d'approximation pour **glouton** trouvé en cherchant parmi les tailles de graphes entre 3 et 100 est 1.5.
- Le pire rapport d'approximation pour **couplage** trouvé en cherchant parmi les tailles de graphes entre 3 et 100 est 2.

Pour conclure, le pire rapport d'approximation que nous avons trouvé expérimentalement pour le couplage est très satisfaisant, car il est en adéquation avec la réalité. En effet, le couplage est avéré être 2-approché. Toutefois, en augmentant le nombre de sommets, couplage se trompe de moins en moins.

Le rapport d'approximation expérimental de glouton quant à lui, varie entre 1 et 1.5. Il a néanmoins tendance à être très faible[11] nous indiquant que glouton se rapproche d'une solution optimale.

[9] entre glouton et couplage.
[10] environ moins de 5
[11] aux alentours de 1.