COMPLEX

Projet: Couverture de graphe

Kevin Meetooa Yassine Bareche

6 novembre 2019

Introduction

0.1 Définitions

Soit un graphe $G = (V, E)$ non orienté, où V est l'ensemble des n sommets et E l'ensemble des m arêtes de G. Une couverture de G est un ensemble V' $\subseteq$ V tel que toute arête e $\in$ E a (au moins) une de ses extrémités dans V.

Le but du problème Vertex Cover est de trouver une couverture contenant un nombre minimum de sommets. Le but de ce projet est d'implémenter différents algorithmes, exacts et approchés, pour résoudre le problème Vertex Cover, et de les tester expérimentalement sur différentes instances. Ce problème est NP-difficile (sa version décision est NP-complète), cela se montre par réduction du problème NP-complet Clique.

Preuve :

Certificat : La couverture du graphe est un certificat. On peut vérifier en temps polynomial qu'un ensemble de sommets V est bien une couverture du graphe en vérifiant par exemple pour chaque arête du graphe l'existence d'un sommet $v \in V$ qui lui est adjacent.

Réduction : L'existence d'une couverture V de taille de k du graphe G est équivalente à l'existence d'une clique de taille $n - k$ dans le graphe complémentaire G^c de G.

$\Rightarrow$ Supposons que G ait une couverture V de taille k. Soient x, y deux sommets de G n'appartenant pas à V. Alors par définition, il n'existe pas d'arête entre x et y, sinon cela signifierait qu'il existe une arête qui n'est pas couverte pas V et V ne serait pas une couverture. Ainsi, on montre que pour tous sommets $x \notin V$, $y \notin V$, x et y ne sont pas adjacents. Par conséquent, on peut construire une clique de taille n-k dans G^c.

$\Leftarrow$ Supposons qu'il y ait une clique C de taille n-k dans G^c. Alors les sommets de C forment un ensemble de sommets 2 à 2 non adjacents dans G. Ainsi, les n-(n-k) sommets de G qui ne sont pas dans C couvrent toutes les arêtes de G par définition du graphe complémentaire. Donc il existe une couverture de taille k de G.

Implémentation

0.2 Choix d'implémentation

Nous avons décidé d'implémenter notre classe Graph en Python. Ce choix est principalement motivé par les facilités d'implémentations offertes par Python dans le cadre de notre problème. En effet, il n'y a pas d'allocation mémoire à gérer comme cela aurait pu être le cas dans d'autres languages (C) et la structure de liste et de dictionnaire en python est particulièrement simple d'utilisation tout en restant adaptée à notre problème. C'est donc une représentation d'un graphe sous forme de liste d'adjacence que nous avons choisi : à chaque noeud, on associe une liste de ses voisins.

Attributs de la classe Graph :
* Un tableau dans lequel on stocke les noeuds du graphe.
* Un dictionnaire associant à chaque noeud du graphe la liste de ses voisins

Méthodes de base définies dans la classe Graph :
* Une fonction renvoyant la taille du graphe (nombre de sommets).
* Des fonctions d'ajout et de suppression de noeuds et d'arêtes
* Des fonctions permettant d'obtenir le degré d'un noeud donné en argument, le noeud de degré maximal du graphe ainsi que le degré associé.
* Une fonction permettant de copier un graphe
* Une fonction prenant en entrée un entier $n > 0$ et un paramètre $p \in]0, 1[$, et qui renvoie un graphe sur n sommets où chaque arête (i, j) est présente avec probabilité p.

0.3 Protocole expérimental

Dans tous les tests expérimentaux que nous présenterons par la suite, nous appliquerons le protocole expérimental suivant :

— Toutes nos fonctions auront au moins 2 attributs *inst* et *inter*, *inst* représentant le nombre d'instances sur lequel nous voulons tester une fonction et *inter* représentant le nombre de fois que nous testerons la fonction pour une instance donnée.

— Pour chaque instance, nous testons *inter* fois la fonction avant de faire une moyenne arithmétique sur le résultat obtenu. Cela permet d'améliorer la précision des tests.

0.4 Premier test expérimental : Génération de graphes aléatoires en fonction du nombre de sommets

Dans notre premier test expérimental, nous traçons la durée d'exécution de notre fonction permettant de générer des graphes aléatoires en fonction du nombre de sommets. Nous traçons plusieurs graphes correspondant à des probabilités d'apparition des arêtes différentes (Nous avons choisi 0.1, 0.5 et 0.9). Voici les résultats obtenus :

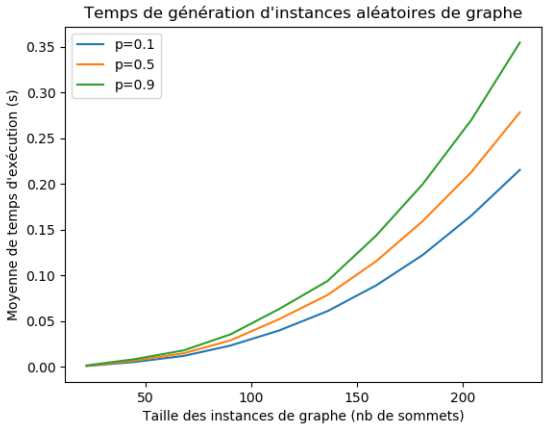

FIGURE 1 – Temps de génération des instances en fonction du nombre de noeuds

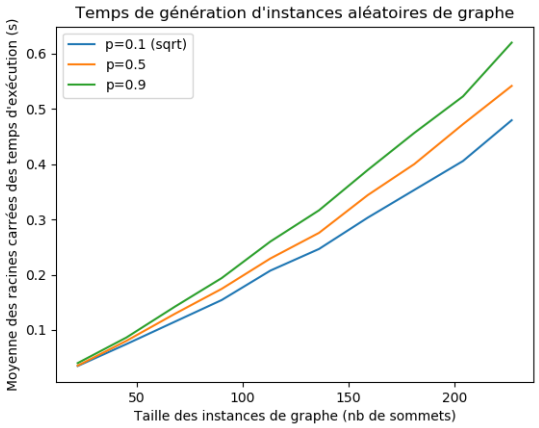

FIGURE 2 – Racine carrée du temps de génération des instances

La figure 1 montre que les graphes générés aléatoirement prennent un temps d'exécution plus important lorsque la probabilité d'apparition des arêtes p augmente, ce qui est cohérent. En effet, plus p est grand et plus il y a d'arêtes à insérer.

De plus, la figure 2 montre que la racine carrée du temps d'exécution peut être approchée avec une grande précision par une fonction linéaire. Ainsi, on peut en déduire que l'algorithme de génération d'instances aléatoires est de complexité $\Theta(n^2)$

Algorithmes

0.5 Premiers algorithmes : Algorithme de couplage et algorithme glouton

0.5.1 Principe

Nous allons commencer par une approche naïve du problème. Une approche naïve possible est de créer un ensemble de sommets vide C puis de regarder toutes les arêtes du graphe : Si l'arête n'est pas couverte par un sommet de C, on ajoute ses deux extrémités à C. On répète ce processus jusqu'à couvrir toutes les arêtes du graphe. Cet algorithme correspond à l'algorithme de couplage donné ci-dessous :

Algorithme 1 Algo couplage

```
C = Ensemble vide
Pour i de 1 à m :
    Si aucune des extrémités de e_i n'est dans C :
        Ajouter les 2 extrémités de e_i à C
    Fin Si
Fin Pour
Renvoyer C
```

Une autre approche naïve possible est de créer un ensemble de sommets vide C puis d'y ajouter le sommet de degré maximal du graphe en supprimant ses arêtes adjacentes dans le graphe. On répète ce processus jusqu'à ce qu'il n'y ait plus d'arêtes dans le graphe. Cet algorithme correspont à l'algorithme glouton donné ci-dessous :

Algorithme 2 Algo glouton

```
C = Ensemble vide
Pour i de 1 à m :
    Tant que E n'est pas vide :
        Prendre un sommet v de degré maximum
        Ajouter v à c, supprimer de E les arêtes couvertes par v
    Fin Si
Fin Pour
Renvoyer C
```

0.5.2 Mesures expérimentales :

Nous avons d'abord comparé ces deux algorithmes en termes de temps d'exécution en fonction du nombre d'arêtes. Nous les avons ensuite comparé en termes de qualité (taille des couvertures renvoyées) en fonction du nombre d'arêtes. Nous avons obtenu les courbes suivantes.

Voici les résultats obtenus lorsque p=0.2 :

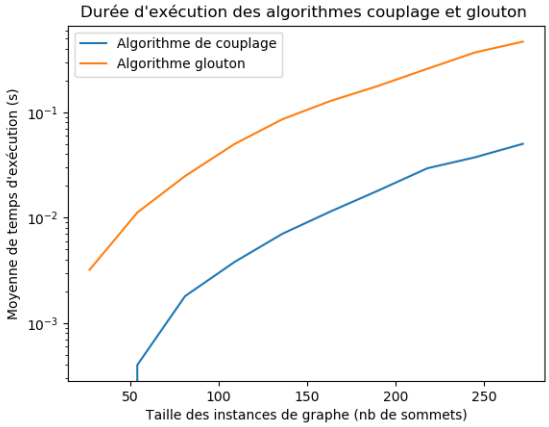

FIGURE 3 – Temps d'exécution en échelle logarithmique en fonction du nombre d'arêtes

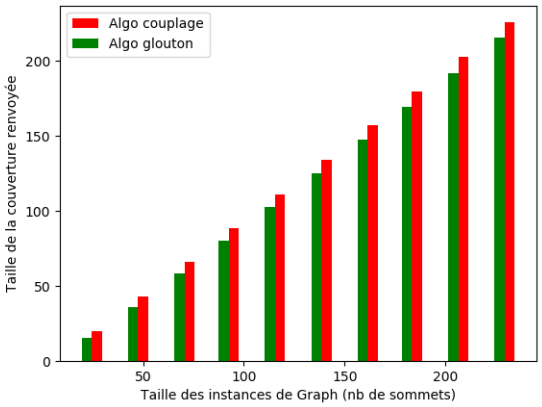

FIGURE 4 – Taille des solutions (longueur de la couverture renvoyée) en fonction du nombre d'arêtes

La figure 3 montre que les deux algorithmes couplage et glouton sont des algorithmes de complexité temporelle exponentielle. On remarque que l'algorithme de couplage est toujours plus rapide que l'algorithme glouton.

La figure 4 montre que l'algorithme glouton renvoie toujours une meilleure solution (en terme de taille) que l'algorithme de couplage.

Ces deux figures ont été obtenues avec une valeur fixe de p=0.2

Nous avons ensuite tracé les mêmes courbes avec une valeur fixe de p=0.8

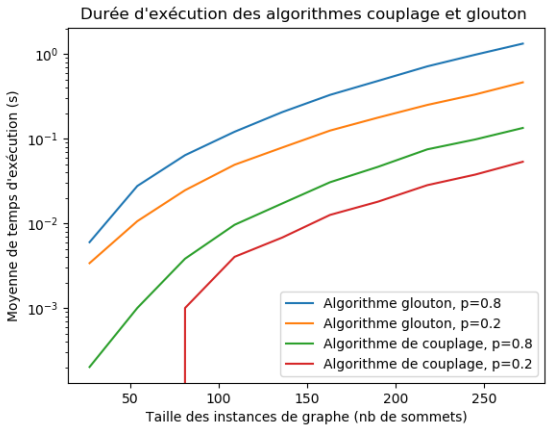

FIGURE 5 – Temps d'exécution en échelle logarithmique en fonction du nombre d'arêtes

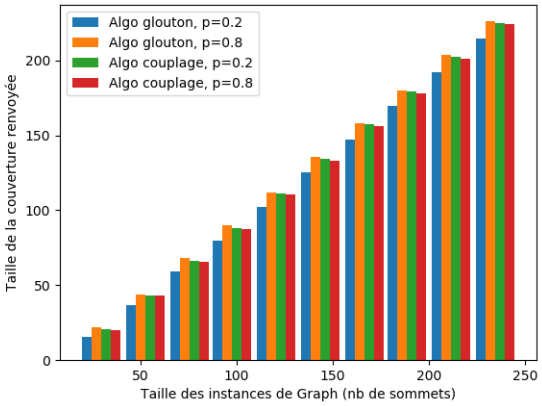

FIGURE 6 – Taille des solutions (longueur de la couverture renvoyée) en fonction du nombre d'arêtes

Comme précédemment, la valeur de p influe les durées d'exécution des algorithmes : La figure 5 montre que pour un graphe donné, plus p est grand, plus les algorithmes prennent du temps, ce qui est cohérent.

De même, la figure 6 montre que pour un graphe donné, plus p est grand, plus la taille des solutions renvoyées par l'algorithme de couplage est petite, ce qui est cohérent car un graphe avec un grand nombre d'arêtes possède une couverture de petite taille.

Cependant, l'algorithme glouton renvoie des solutions de tailles similaires indépendamment de p.

Ces deux algorithmes ne sont pas optimaux. L'algorithme de couplage est 2-approché tandis que l'algorithme glouton n'est pas r-approché pour r aussi grand que possible.

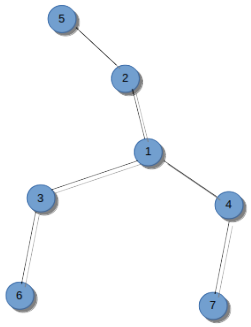

FIGURE 7 – Exemple montrant que l'algorithme glouton n'est pas optimal

La figure 7 montre que l'algorithme glouton n'est pas optimal. En effet, sur cette instance, l'algorithme glouton retourne l'ensemble {1,2,3,4} alors que la solution optimale est {2,3,4}

0.6 Algorithmes de séparation et évaluation

0.6.1 Branchement

Dans cette partie, on se propose d'implémenter un algorithme de branchement simple. Le branchement considéré ici est le suivant : prendre une arête e = {u, v}, et considérer deux cas : soit u est dans la couverture, soit v est dans la couverture.

Ainsi, partant d'un graphe initial G et d'un ensemble de sommets C vide représentant la solution, la première branche consiste à mettre u dans C et à raisonner sur le graphe où l'on a supprimé u (les arêtes incidentes à u sont couvertes) ; symétriquement, la deuxième consiste à mettre v dans C et à raisonner sur le graphe où l'on a supprimé v.

Nous avons implémenté cet algorithme de façon récursive puis nous avons cherché à évaluer son efficacité en fonction de n et p.

Pour cela, nous avons tracé le temps d'exécution de l'algorithme de branchement en fonction du nombre de sommets de graphes générés aléatoirement. Nous avons obtenus les résultats suivants pour p=0.2 et p=0.8

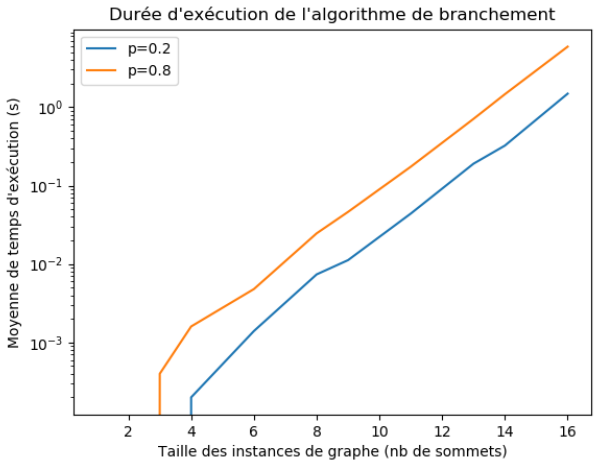

FIGURE 8 – Temps d'exécution de l'algorithme de branchement en fonction du nombre de sommets pour différentes valeurs de p

La figure 8 montre que l'algorithme est de complexité temporelle exponentielle et que lorsque p augmente, le temps d'exécution augmente ce qui est cohérent.

Dans toute la suite, on prendra $p=\frac{1}{\sqrt{n}}$ où n est la taille de l'instance. Cela nous permettra d'avoir des graphes moins denses afin de faciliter les tests expérimentaux.

0.6.2 Ajout de bornes

On cherche maintenant une borne inférieure de notre graphe G afin d'avoir un critère de séparation nous permettant de gagner en complexité temporelle.

Soit M un couplage de G et C une couverture de G.
Posons b_1=$\lceil \frac{m}{\Delta} \rceil$ (avec Δ le degré maximum des sommets du graphe).
b_2=|M|
b_3=$\frac{2n-1-\sqrt{(2n-1)^2-8m}}{2}$
Montrons que |C| $\geq$ max{b_1,b_2,b_3}

— Borne b_1 : Par définition, toutes les arêtes de G sont couvertes par C $\forall e \in M, \exists v \in C$, e est adjacente à v
De plus, m $\leq \sum_{v \in C} deg(v)$
Or, deg(v) $\leq \Delta$
Donc m $\leq \sum_{v \in C} \Delta$
m $\leq$ |C|Δ
$\Rightarrow$ |C| $\geq \lceil \frac{m}{\Delta} \rceil$ (la partie entière vient du fait que |C| $\in \mathbb{N}$

— Borne b_2 : Par définition, une couverture doit couvrir toute arête du graphe. En particulier, toutes les arêtes de M doivent être couvertes par les sommets de C.
$\forall e \in M, \exists v \in C$, e est adjacente à v
Ainsi, $\forall e1, e2 \in M$ avec $e1 \neq e2$, alors $\exists v1, v2 \in C$: e1 est adjacente à v1 et e2 est adjacente à v2.
De plus, $e1$ et $e2$ étant distinctes, elles ne partagent aucun sommet en commun car M est un couplage de G. Par conséquent, $e1 \neq e2$
On en déduit que pour toute arête du couplage M, il existe au moins un sommet de la couverture C qui lui est adjacent.
Donc |C| $\geq$ |M|

— Borne b_3 : On pose P(X)=X^2+(1-2n)X+2m
P est un polynôme de degré 2 de coefficient dominant positif.
P(|C|)=$|C|^2$+(1-2n)|C|+2m
De plus, |C| $\leq n$ et $m \leq \frac{n(n-1)}{2}$
Ainsi, P(|C|) $\leq n^2 + (1-2n)n + 2\frac{n(n-1)}{2}$
P(|C|) $\leq n^2 + n - 2n^2 + n^2 + n = 0$
Donc P(|C|) ≤ 0
$\Leftrightarrow$ |C| $\in [r_1, r_2]$ où r_1 et r_2 sont les deux racines du polynôme P.
r_1=$\frac{2n-1-\sqrt{(2n-1)^2-8m}}{2}$ = b_3 et r_2=$\frac{2n-1+\sqrt{(2n-1)^2-8m}}{2}$
Ainsi, on a bien |C| $\geq b_3$

On en déduit que |C| $\geq$ max{b_1,b_2,b_3}

Nous avons ensuite inséré le calcul de cette borne inférieure dans notre algorithme de branchement et nous l'avons comparé à la version sans bornes.

La figure 9 montre que l'ajout de la borne inférieure nous fait bien gagner en complexité temporelle.

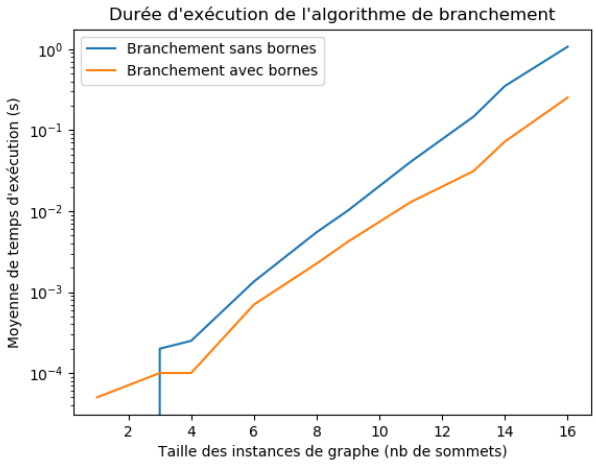

FIGURE 9 – Temps d'exécution de l'algorithme de branchement avec et sans la borne inférieure

0.6.3 Amélioration du branchement

Lorsque l'on branche sur une arête e = {u, v}, dans la deuxième branche où l'on prend le sommet v dans la couverture, on peut supposer que l'on ne prend pas le sommet u (le cas où on le prend étant traité dans la première branche. Dans la 2ème branche, ne prenant pas u dans la couverture on doit alors prendre tous les voisins de u (et on peut les supprimer du graphe donc).

Nous avons donc ajouté cette modification à l'algorithme et avons comparé ce dernier à l'algorithme précédent. Cela devrait nous faire gagner en complexité temporelle. Voici les résultats obtenus.

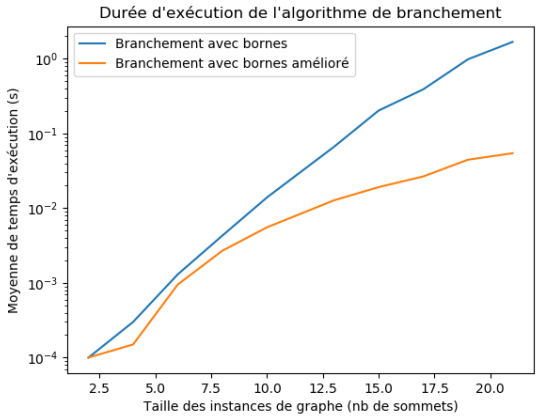

FIGURE 10 – Temps d'exécution de l'algorithme de branchement avec et sans amélioration

Comme attendu, la figure 10 montre que cette modification nous fait gagner en complexité temporelle. De plus, l'algorithme de branchement avec bornes nous permettait de traiter des graphes d'environ 18 noeuds en une seconde. Cette version améliorée nous permet de traiter des graphes d'environ 24 noeuds en une seconde.

Il est cependant encore possible d'améliorer le branchement. En effet, afin d'éliminer un maximum de sommets dans la deuxième branche, il semble intéressant de choisir le branchement de manière à ce que le sommet u soit de degré maximum dans le graphe restant.

De plus, si un sommet u est de degré 1, il existe toujours une couverture optimale qui ne contient pas u. En effet, prenons C une couverture minimale contenant u. u étant de degré 1, il possède un unique voisin v. En remplaçant u par v dans la couverture C, on garde toujours une couverture optimale.

Nous avons inséré ces simples modifications dans l'algorithme précédent et avons comparé ce dernier au nouvel algorithme ainsi obtenu. Voici les résultats que nous avons obtenu.

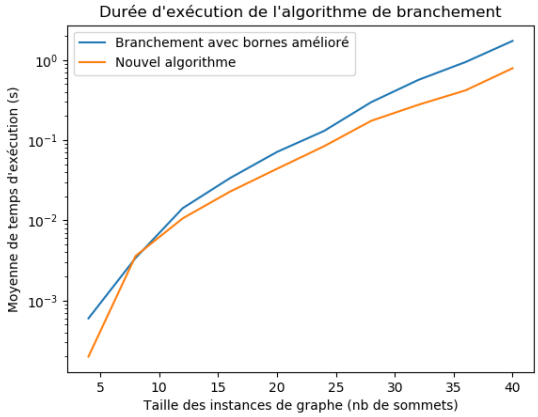

FIGURE 11 – Temps d'exécution de l'algorithme de branchement

La figure 11 montre que cette modification a permis de gagner en complexité temporelle même si le gain est minime.

On remarque que le nombre de noeuds traités en environ une seconde est maintenant d'environ 45 soit plus du double de ce qui était traitable avec l'algorithme de départ (branchement sans bornes) pour une durée d'exécution similaire.

Conclusion

Au cours de ce projet, nous avons commencé par des algorithmes naïfs renvoyant des solutions non optimales au problème de couverture de graphe.

Nous avons ensuite implémenté un algorithme de branchement simple que nous avons amélioré en y ajoutant des critères de séparation.

Cependant, le problème de couverture de graphe est un problème NP-difficile et n'est pas résolvable en temps polynomial. Ainsi, tous les algorithmes que nous avons implémentés sont de complexité exponentielle. Des critères de séparation plus ou moins efficaces peuvent améliorer la complexité de l'algorithme mais celui-ci restera toujours exponentiel et explosera lorsque la taille des instances augmentera.