Projet COMPLEX

Benjamin Tordjman 21305112
Cyrus Mansour 21304236

October 22, 2023

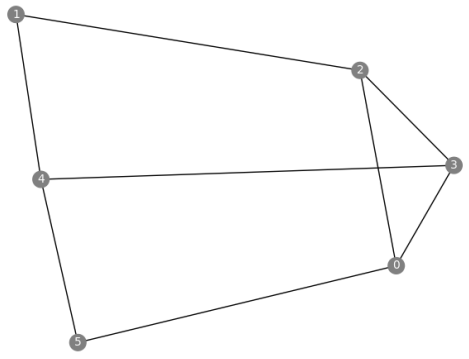
1 Introduction

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. Ce problème est NP-difficile (sa version décision est NP-complète). 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.

2 Graphes

L'implémentation de ce projet est faite en Python, et les graphes sont implémentés grace à la bibliothèque NetworkX.
Il est possible de créer un graphe à partir d'un fichier texte grace à la fonction $loadGraph(filename)$, le format de ce fichier doit être identique à celui du fichier "exempleinstance.txt".

Graphe décrit dans le fichier instance.txt:

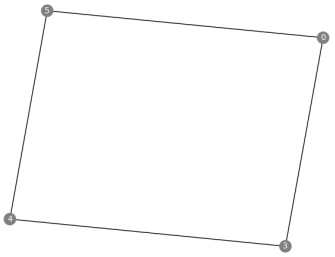


2.1 Opérations de base

1. $deleteVertex(g, v)$
   Retourne une copie du graphe g où l'on a supprimé le sommet v.

2. $deleteVertices(g, vertices)$
   Retourne une copie du graphe g où l'on a supprimé un ensemble de sommets contenus dans la liste $vertices$.

Exemple où l'on supprime les sommets 1 et 2:

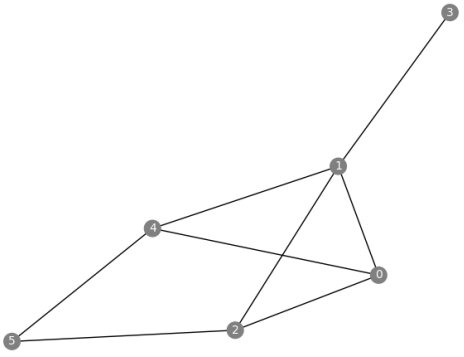


3. $getVerticesDegrees(g)$
   Retourne un dictionnaire qui associe chaque sommet du graphe g à son degré (le nombre d'arêtes qu'il couvre).
   $getMaxDegreeVertex(g)$
   Retourne un sommet de g ayant le degré le plus élevé.

2.2 Génération d'instances

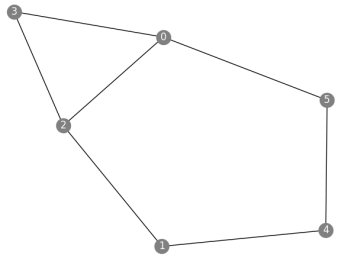
La fonction $generateRandomGraph(n, p)$ retourne un graphe suivant la loi de probabilité Erdös-Rényi: il s'agit d'un graphe ayant n sommets et pour lequel chaque arête possible a une probabilité p d'exister.

Graphe généré par cette fonction avec $n = 6$ et $p = 0.3$:

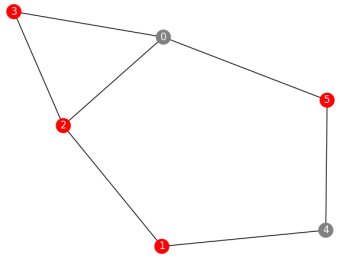


3 Méthodes approchées

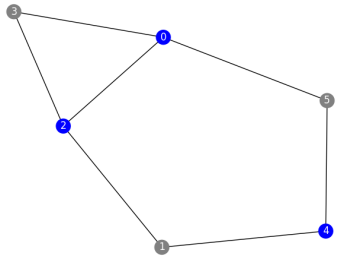
1. L'algorithme glouton, lorsqu'appliqué à l'instance suivante:



retourne cette couverture:



Cependant, une couverture optimale serait:

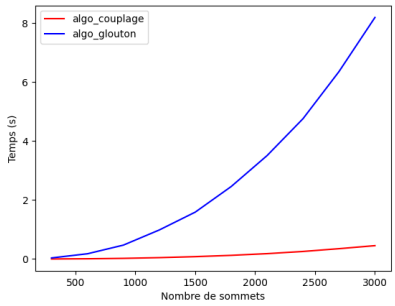


Donc l'algorithme glouton n'est **pas optimal**.
De plus, nous pouvons affirmer qu'il n'est **pas r-approché pour r** $< \frac{4}{3}$.

2. Rappel: n représente le nombre de sommets, p représente la probabilité d'existence de chaque arête possible du graphe G.

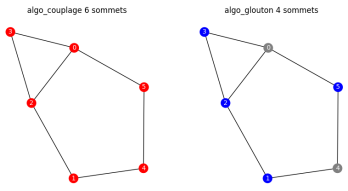
   (i) Temps de calcul:

   - **algo_couplage**: parcourt les arêtes du graphe une par une, ce qui signifie que le temps de calcul dépend linéairement du nombre d'arêtes dans le graphe. Sa complexité temporelle dans le pire des cas est de l'ordre de $O(m)$, avec m le nombre d'arêtes.
   - **algo_glouton**: sélectionne un sommet de degré maximum à chaque itération, puis supprime les arêtes couvertes par ce sommet. Le temps de calcul dépend donc principalement du nombre de sommets et du degré de chaque sommet. Dans le pire des cas, chaque sommet est de degré 1, et donc sa complexité devient $O(m * n)$ où m est le nombre d'arêtes et n le nombre de sommets (on multiplie par n car la fonction utilisée pour trouver l'arête de degré maximum est en $O(n)$).



   (ii) Qualité des solutions retournées:

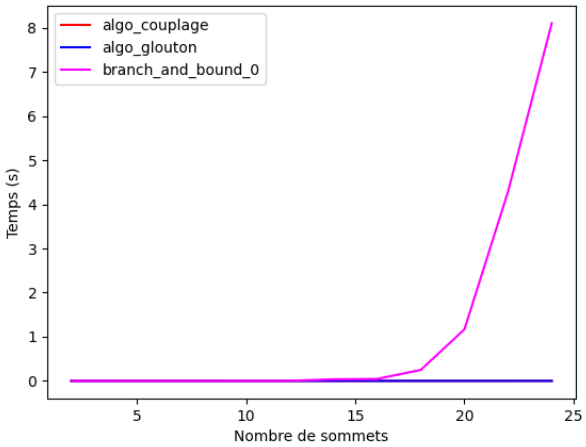
   - **algo_couplage**: garantit une couverture, mais la qualité de la couverture est de plus en plus médiocre au fur et à mesure que n croît. Il est assez fréquent que cet algorithme choisisse des sommets évidemment inutiles.
   - **algo_glouton**: tend à donner de meilleures couvertures en sélectionnant les sommets de degré maximum et en évitant les sommets inutiles. Cependant, cela ne garantit pas toujours une couverture optimale, car il ne fait que des choix localement optimaux.



4 Séparation et évaluation

4.1 Branchement

La méthode *branch_and_bound_0* visite tous les sous-ensembles possibles de l'ensemble des sommets. Il n'est pas étonnant que l'algorithme s'exécute de plus en plus lentement au fur et à mesure que n croît: sa complexité temporelle est exponentielle. L'algorithme ne s'arrête lorsqu'il trouve la solution optimale, il continue donc son exécution jusqu'à avoir parcouru l'intégralité de l'arbre. p étant la probabilité qu'une arête existe, plus p est élevé, plus il y a d'arêtes, et comme le nombre d'opérations de l'algorithme est proportionnel au nombre d'arêtes, alors le temps d'exécution de l'algorithme est proportionnel à p.



On voit bien que sur un graphe de taille $n = 25$ l'algorithme *branch_and_bound_0* prend 8 secondes pour retourner une solution, alors que les autres en retourne une quasiment instantanément.

4.2 Ajout de bornes

1. En considérant les données et notations de l'énoncé, montrons la validité de chacune des bornes b_1, b_2, b_3:

   – **b_1**:
   La borne b_1 est basée sur le degré maximum des sommets du graphe.
   On veut montrer $|C| \geq b_1$.
   Supposons $|C| < b_1$, cela signifie que la taille de la couverture est insuffisante pour couvrir tous les sommets: le nombre d'arêtes couvertes par la couverture, m, est inférieur au produit du degré maximum des sommets et de la taille de la couverture, $b_1 \cdot |C|$.
   Cela implique qu'il y a au moins une arête non couverte, ce qui contredit la définition de C en tant que couverture.

Donc, $|C| \geq b_1$.
La borne $\mathbf{b_1}$ est donc valide.

– **$\mathbf{b_2}$**:
La borne b_2 est simplement $|M|$.
On veut montrer $|C| \geq b_2$.
Comme C est une couverture, elle doit couvrir toutes les arêtes du couplage M. Cela implique que $|C|$ doit être au moins égal à $|M|$.
Donc, $|C| \geq b_2$.
La borne $\mathbf{b_2}$ est donc valide.

– **$\mathbf{b_3}$**:
La borne b_3 dépend du nombre maximal d'arêtes dans un graphe dont C est une couverture. On peut montrer la validité de b_3 en s'interrogeant sur ce nombre maximal. Cela se produit lorsque la couverture est minimale, c'est-à-dire lorsque tous les sommets dans C sont essentiels pour couvrir les arêtes.
Soient m le nombre d'arêtes dans G, n le nombre de sommets dans G, et C une couverture de G.
Dans un graphe complet, le nombre d'arêtes totales est:

$$\frac{n \times (n-1)}{2}$$

Maintenant, le nombre d'arêtes entre les sommets qui n'appartiennent pas à la couverture C est:

$$\frac{(n-|C|) \times (n-|C|-1)}{2}$$

Ainsi, le nombre d'arêtes totales dans C est la différence des deux:

$$\begin{aligned}
&\frac{1}{2}\Big(n \times (n-1) - (n-|C|) \times (n-|C|-1)\Big) \\
&= \frac{1}{2}\Big(n^2 - n - n^2 + n|C| - |C|^2 - |C|\Big) \\
&= \frac{1}{2}\Big(2n|C| - |C|^2 - |C|\Big)
\end{aligned}$$

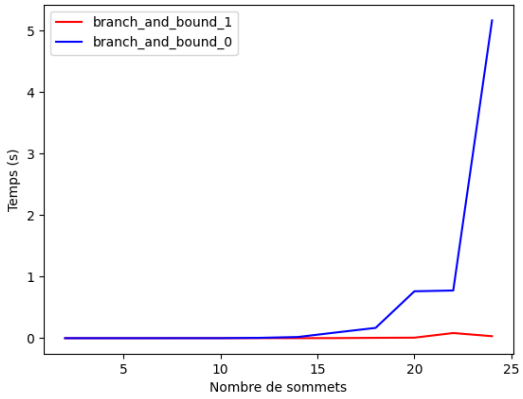
On a ainsi:

$$\begin{aligned}
&\frac{1}{2}\Big(2n|C| - |C|^2 - |C|\Big) = m \\
\Leftrightarrow\ &\frac{1}{2}\Big(2n|C| - |C|^2 - |C|\Big) - m = 0 \\
\Leftrightarrow\ &-\frac{1}{2}|C|^2 + \frac{2n-1}{2}|C| - m = 0 \\
\Leftrightarrow\ &|C| = \frac{\frac{-2n+1}{2} \pm \sqrt{\left(\frac{2n-1}{2}\right)^2 - \frac{4}{2}m}}{2}
\end{aligned}$$

$$\Leftrightarrow |C| = \frac{2n - 1 \pm \sqrt{(2n-1)^2 - 8m}}{2}$$

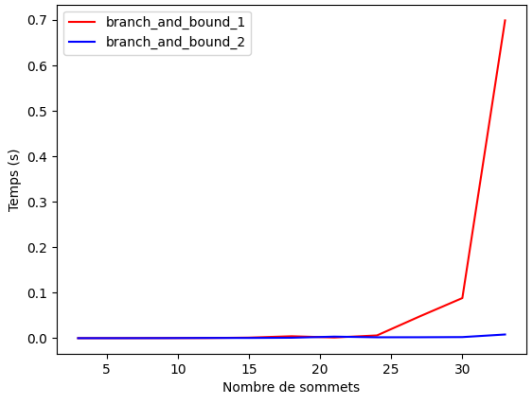
On reconnaît la définition de b_3 de l'énoncé et on constate que $|C| = b_3$ dans le cas où la racine est négative.
Donc, $|C| \geq b_3$.
La borne b_3 est donc valide.

2. L'algorithme de branchement modifié, appelé *branch_and_bound_1*, implémente les bornes b_1, b_2 et b_3. Cela nous permet de visiter moins de nœuds en élaguant les branches qui n'aboutissent pas à une solution optimale.
   Le graphique suivant montre à quel point cette optimisation réduit le temps de calcul:

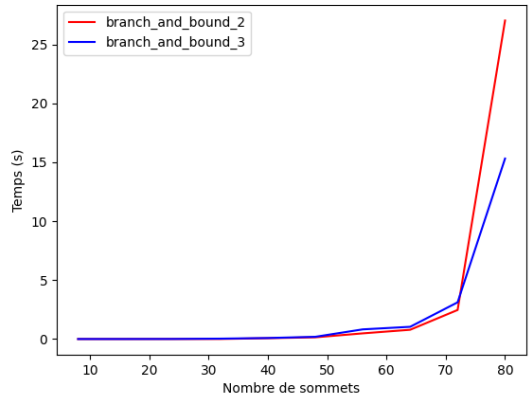


4.3 Amélioration du branchement

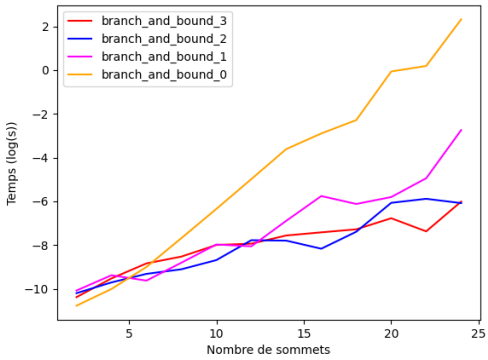
1. L'algorithme de branchement amélioré, appelé *branch_and_bound_2*, nous permet de retirer $(k-1)$ branchements, où k est le degré de u.
   En effet, dans cette méthode, si l'on visite le sommet v, un voisin de u, on ne visite pas u, mais on ajoute à la couverture actuelle tous les autres voisins de u.
   Le graphique suivant nous montre à quel point cela améliore notre temps de calcul:

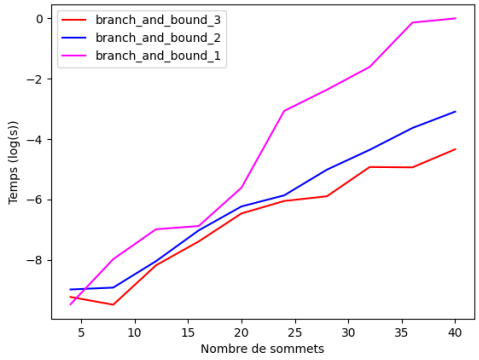


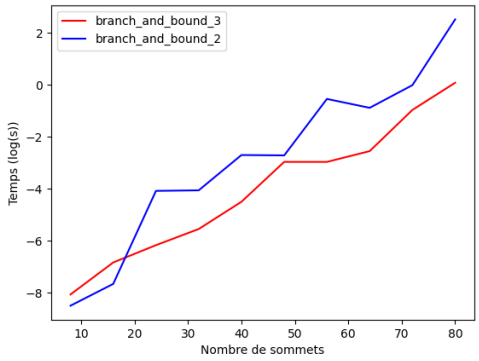
2. Dans l'algorithme précédent, où l'on évite $(k-1)$ branchements, il serait plus avantageux de maximiser k pour reduire davantage le temps de calcul. C'est pourquoi dans l'algorithme *branch_and_bound_3*, on prend u étant le sommet de degré max.
   Le graphique suivant nous montre une légère amélioration:



En somme, les trois graphiques suivants comparent, sur une échelle logarithmique, les différentes méthodes de branchement.







Notez que n monte jusqu'à 80 pour les deux derniers algorithmes car ils sont capables de trouver une couverture en un temps raisonnable. En comparaison, l'algorithme branch_and_bound_0 initial, ne va pas au delà de $n = 30$.

4.4 Qualité des algorithmes approchés

1. Une évaluation expérimentale nous donne les rapports d'approximation empiriques pour chacun des deux algorithmes:

```
algo_couplage:
        n = 10, moyenne = 1.62, pire = 2.0
        n = 20, moyenne = 1.5, pire = 1.8
        n = 30, moyenne = 1.46, pire = 1.75
        n = 40, moyenne = 1.44, pire = 1.67
        n = 50, moyenne = 1.41, pire = 1.6
algo_glouton:
        n = 10, moyenne = 1.02, pire = 1.25
        n = 20, moyenne = 1.04, pire = 1.18
        n = 30, moyenne = 1.04, pire = 1.18
        n = 40, moyenne = 1.05, pire = 1.16
        n = 50, moyenne = 1.05, pire = 1.17
```

Nous pouvons donc en conclure que l'algorithme glouton est relativement proche de l'optimale, tout en s'exécutant considérablement plus vite.