Samba Diarra Diouf

Mohamed Irazzi

Projet d'algorithme des graphes

Le but de ce projet est de proposer et implémenter un algorithme heuristique proposant une solution réalisable pour ce problème. C'est à dire qu'il n'est pas demandé d'avoir une solution forcement *optimale*, mais une solution *valide* : l'algorithme doit forcément produire une union de cliques au final.

Afin de résoudre ce problème, nous avons directement pensé à regarder dans les documents fournis par le professeur. Nous avons lu le document *"Efficient algorithms for cluster editing*", de Lucas Bastos · Luiz Satoru Ochi · Fábio Protti ·Anand Subramanian · Ivan César Martins ·Rian Gabriel S. Pinheiro. Lors de la lecture de celui-ci, nous avons constaté qu'il y avait deux solutions pouvant résoudre notre problème, l'algorithme de GRASP et l'algorithme d'ILS. Nous avons opté pour l'algorithme de GRASP, que nous avions le mieux compris.

GRASP est une métaheuristique pour l'optimisation de problèmes, dans lesquels chaque itération se compose essentiellement de deux phases : la construction et la recherche locale. La phase de construction construit une solution réalisable, dont les voisins sont explorés jusqu'à ce qu'un optimum local soit trouvé pendant la phase de recherche locale. Le meilleur résultat global est stocké en tant que solution de notre problème.

Maintenant, nous allons expliquer les grandes lignes de l'algorithme permettant de trouver la solution optimale à notre problème.

Tout d'abord nous avons créé une fonction grasp(G,Tmax) qui prend en paramètre un graphe de départ G (implémenté sous forme d'un dictionnaire) et un temps Tmax, qui lui représente la limite de temps dans laquelle notre algorithme peut tourner. Dans cette fonction, initialement une solution G1 est générée et définie comme étant la meilleure solution en appliquant la phase de construction. Ensuite une variable Tstart est initialisée avec la fonction time() qui renvoie l'heure actuelle. Puis tant que le temps Tmax n'a pas été atteint, une nouvelle solution G2 est générée avec la phase de construction. Sur cette solution G2 générée, on essaye d'appliquer la phase de Local Search pour optimiser la solution et une fois le Local Seach terminé, on vérifie si le coût de la solution G2 obtenu est inférieur à celui de la solution G1. Si oui, G2 devient la solution optimale et G1 prend la valeur de G2. Une fois le temps Tmax atteint, on renvoie la dernière solution optimale obtenue. Complexité : $O(tmax\ n^2)$

Phase de construction

La phase de construction consiste à générer une solution valable à notre problème. Il faut savoir qu'il existe deux types de construction : Relative neighborhood et Vertex agglomeration. Nous avons choisi d'implémenter le Relative neighborhood.

Pour se faire, dans notre fonction construction(G) qui prend en paramètre un graphe G(implémenté sous forme de dictionnaire), nous initialisons une variable Kbest qui aura une valeur choisie au hasard entre 1 et n (le nombre de sommets du graphe). Ensuite une variable K représente le cluster de notre solution est initialisée, K aura une valeur choisie au hasard entre Kmin et Kmax où Kmin=max(Kbest – racine(n), 1) et Kmax = min(Kbest + racine(n), n). Ensuite nous récupérons et stockons le nombre de voisins de chaque sommet dans un tableau heapq. Nous avons choisi un tableau heapq car cela car nous de récupérer de façon décroissante la liste des sommets ayant le plus grand nombre de voisins plus facilement. Puis on choisit les K premiers éléments de V (liste contenant tous les sommets du graphe) dont le nombre de voisins est le plus grand. Ainsi chaque sommet sélectionné représentera le premier élément d'un nouveau groupe de cluster et est retiré de V. Ensuite tant que V n'est pas vide, on lui retire un sommet afin de l'ajouter dans des K cluster créés. Le cluster qui maximise RN(node,Ci) sera le cluster où on rajoutera le sommet node. RN(node,Ci) représente la différence entre le nombre de sommet avec qui le sommet node est voisin dans Ci et le nombre de sommet avec qui node n'est pas voisin dans Ci.
S'il existe un sommet node dont d(node)=0 alors node sera dans un nouveau cluster à lui seul.

Une fois que V est vide, c'est-à-dire l'affectation de sommet à un cluster terminée, toutes les arêtes reliant des sommets qui ne sont pas dans le même cluster sont supprimées. Puis on lie tous les sommets d'un cluster de façon à former un ensemble de cliques. Pour finir la fonction renvoie le graphe et la liste de cluster obtenue.

Complexité : $O(n^2)$

Local Search

Une fois la phase de construction terminée, celle du local search commence.
Le local Search vise à trouver une solution plus optimale que celle trouvée dans la phase de construction.

Tout d'abord nous avions codé qu'un seul algorithme de local search, le empty cluster. Par la suite on a remarqué que sur certains graphes l'algorithme du cluster split donnait parfois une solution avec un meilleur coût (moins de modifications d'arrêtes). Nous avons donc décidé d'implémenter les deux.

Premièrement on fait un empty cluster, on stocke le coût de la solution S1 obtenue, et sur cette même solution, on applique un cluster split pour obtenir une solution S2, on compare le coût de S1 et de S2 et celle qui a le meilleur coût, sera celle qui sera choisie.

Empty cluster

L'algorithme du Empty cluster consiste à trouver un cluster C qui maximise la somme des cost+(i, V(G) \ C) pour tout i appartenant à C où cost+(i, V(G) \ C) représente le nombre de sommets qui sont dans d'autres cluster et avec qui i n'est pas voisin. Une fois le cluster C est trouvé, chaque sommet de C sera transféré dans autre Cluster existant C' de façon à ce que ce cluster C' maximise RN(i,C') où RN(i,C') représente la différence entre le nombre de sommet avec qui i est voisin dans C' et le nombre de sommet avec qui i n'est pas voisin dans C'. Une fois tous les sommets de C transférés, il sera supprimé.

Complexité : $O(n^2)$

Cluster Split

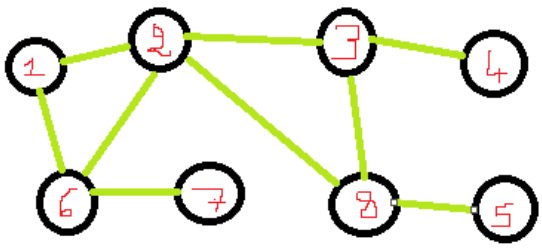
Après avoir fait le Empty cluster, on applique l'algorithme du cluster split sur la solution obtenue afin d'essayer de trouver une solution plus optimale. Comme pour l'algorithme du Empty cluster, on essaye d'abord de trouver un cluster C qui maximise la somme des cost+(i, V(G) \ C) pour tout i appartenant à C. Une fois C trouvé, on sélectionne deux sommets i et j de C qui ne sont pas voisins et dont cost+(i) + cos+(j) est le plus grand dans C. Ces deux sommets représenteront le premier sommet de deux nouveaux clusters. Les sommets restants de C seront transférés en fonction du nombre de voisin en commun soit dans le cluster où se trouve le sommet i soit le cluster où se trouve le sommet j

Complexité : $O(n^2)$

Exemple 1 :

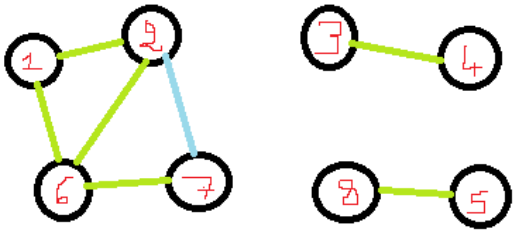
Graphe d'entrée

```
p cep 8 9
1 2
1 6
1 7
2 8
2 6
6 7
3 8
3 4
8 5
```



Graphe de sortie

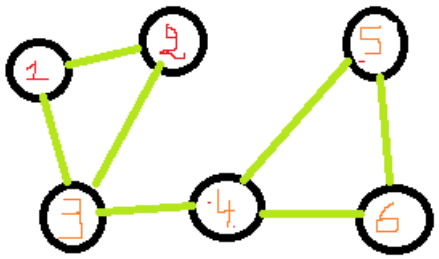
```
2 7
8 3
2 8
```



Exemple 2

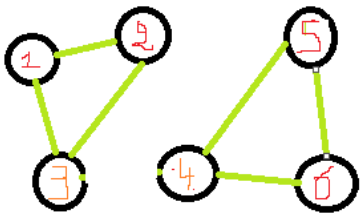
Graphe d'entrée

```
c 2 triangles with an edge in between
c optimal is 1
p cep 6 7
1 2
c bla bla
2 3
3 1
3 4
4 5
5 6
6 4
```



Graphe de sortie

```
3 4
```



Sources

https://www.lamsade.dauphine.fr/~sikora/ens/graphes/projet2020/Paper1.pdf

https://github.com/danielgribel/grasp-clustering/blob/master/grasp.py

https://fr.wikipedia.org/wiki/Greedy_randomized_adaptive_search_procedure