ALASSAF Mohamed

IBOUDA El Mahdi

Dauphine | PSL
UNIVERSITÉ PARIS

Cluster Editing
(Projet Programmation L3 MIAGE 2020-2021)

Introduction

Le but de ce projet est de trouver une solution au problème de Cluster Editing, c'est à dire transformer un graphe (non-orienté) en une union disjointe de cliques en utilisant un nombre minimum de modifications des arêtes (ajouts/suppressions).

Afin de résoudre ce problème nous avons utiliser différents heuristiques et algorithmes qui nous permettent d'obtenir des solutions plus ou moins optimales.

Nous avons décider de développer notre programme en PYTHON.

Nos Approches et algorithmes

Pour bien débuter notre projet et assimiler le sujet, nous avons programmer les deux premiers heuristiques proposés dans le sujet, ces heuristiques sont assez simples mais ils ne sont pas optimaux. Ils consistent à :

1. Supprimer toutes les arêtes du graphe : le graphe résultat est une collection de sommets isolés.
2. Ajouter toutes les arêtes manquantes du graphe : le graphe résultat est une clique.

Par la suite, nous avons chercher à savoir si nous allions appliquer un des heuristiques proposés dans la documentation liée au sujet, ou bien si nous allions créer notre propre algorithme heuristique.

Après plusieurs recherches et lectures d'articles, et en prenant en compte la contrainte de temps d'exécution du programme (10 min maximum), nous avons décidé de programmer un des algorithmes heuristiques proposés dans la littérature (article section 6).

L'algorithme heuristique choisi est l'algorithme GRASP.

Algorithm 1 GRASP pseudocode

```
1: procedure GRASP(G, t_max)
2:     G* ← construction(G)
3:     t_start ← time()
4:     while time() − time_start < time_max do
5:         G' ← construction(G)
6:         G' ← LocalSearch(G')
7:         if s(G') < s(G*) then
8:             G* ← G'
9:         end if
10:    end while
11:    return G* as output
12: end procedure
```

Cet algorithme implémente un heuristique et une recherche locale. La phase de recherche locale « LocalSearch(G') » consiste à tenter d'améliorer la solution renvoyer par la fonction de l'heuristique « construction(G) ».

En ce qui concerne l'heuristique, deux heuristiques étaient proposés, Relative neighborhood et Vertex agglomeration. Nous avons choisi d'implémenter les deux heuristiques.

Pour ce qui est de la recherche locale, trois méthodes étaient proposées, Cluster Split, Empty Cluster et Vertex Move. Nous en avons développé deux : Cluster Split et Empty Cluster.

A chaque itération, notre algorithme va lancer aléatoirement un des deux heuristiques et il lancera également de manière aléatoire une des deux recherches locales.

Leurs Complexités

Pour les 2 heuristiques non optimaux :

1. Supprimer toutes les arêtes du graphe : O(1)
2. Ajouter toutes les arêtes manquantes du graphe : $O(n^2)$

GRASP :

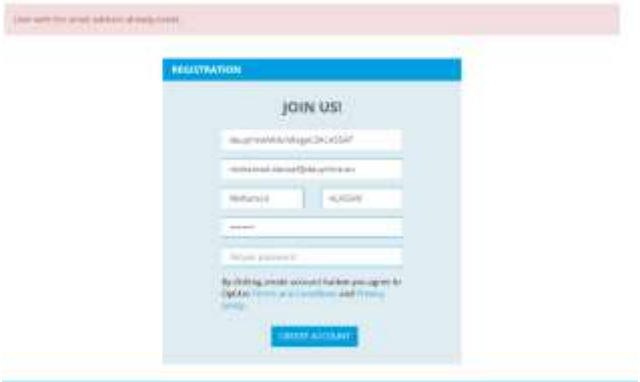
- Les deux heuristiques, pour la fonction « Construction(G) » :
  - Relative neighborhood : $O(n^2)$
  - Vertex agglomeration : $O(n^2)$
- Les deux recherches locales, pour la fonction « LocalSearch(G') » :
  - Cluster Split : $O(n^2)$
  - Empty Cluster : $O(n^2)$

En théorie, notre programme est en complexité $O(n^2)$, étant donné que nous n'avons jamais fait plus de deux boucles imbriquées.

Login site optil.io

Nous avons essayé de nous inscrire sur la plateforme Optil.io, mais nous n'avons jamais reçu de mail de confirmation. Lorsque nous essayons de nous connecter, nous avons ces erreurs.

Et lorsque nous essayons de nous réinscrire, le site nous informe que nos identifiants sont déjà existants dans la base de données.

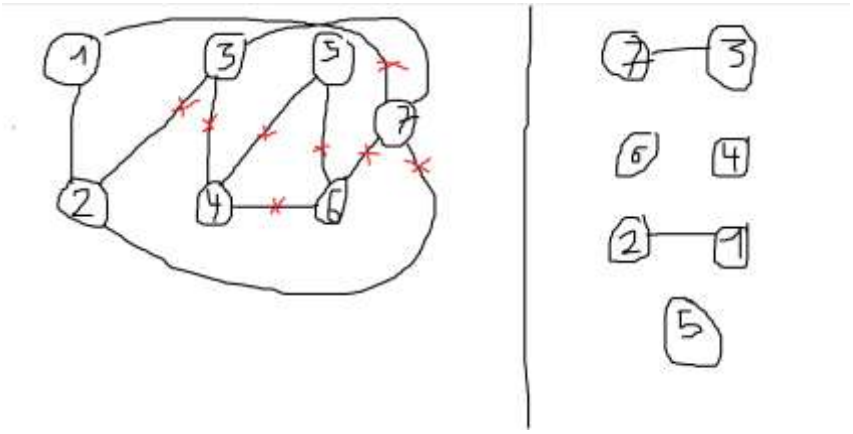


Graphes de tests

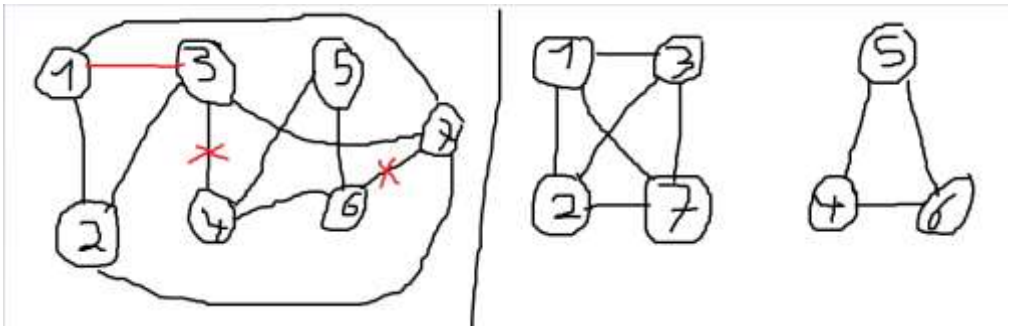
Instance ex.gr :

Entrée :

```
c optimal solution is 3
p cep 7 10
1 2
2 3
3 4
4 5
5 6
6 7
7 1
2 7
3 7
4 6
```



Avant l'optimisation de notre programme nous trouvions une solution pour ce graphe via 8 suppressions d'arêtes.



Sortie :

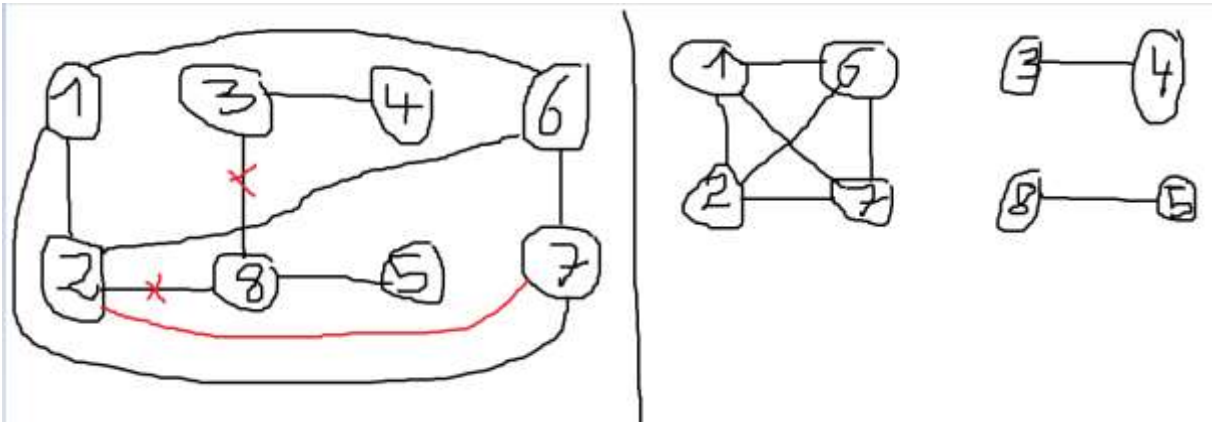
Après l'amélioration de notre programme nous avons obtenu une meilleure solution, en 3 ajouts/suppressions d'arêtes. Cette solution est plus optimale que la précédente et c'est également la plus optimale.

Instance bastos.gr :

Entrée :

```
p cep 8 9
c optimal solution of size 3
1 2
1 6
1 7
6 7
2 6
2 8
3 8
3 4
8 5
```

Sortie :



Après la finalisation de notre programme, nous avons obtenu un résultat en 3 ajouts/suppressions d'arêtes pour ce graphe, ce résultat est bien optimal, comme on peut le voir dans les commentaires au niveau de l'instance d'entrée.

Instance triangle.gr :

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
```

Sortie :

```
4 3
```

Comme on peut le voir en sortie, nous obtenons la solution optimale avec notre programme.

Instance webpage.gr :

Entrée :

```
p cep 7 10
c optimal is 3 editions
1 2
2 3
3 4
4 1
2 4
2 5
5 6
6 7
7 5
7 4
```

Sortie :

```
3 1
7 4
5 2
```

Comme on peut le voir en sortie, nous obtenons la solution optimale avec notre programme.

NB : L'algorithme fonctionne avec des jeux de données avec plusieurs millions d'arêtes en entrée (par exemple le JDD heur161.gr). Cependant lorsque le graphe comporte plusieurs dizaines de milliers de sommet, l'algorithme peine à donner une solution en moins de 10 minutes.

Sources utilisées

Les deux premiers heuristiques :

https://www.lamsade.dauphine.fr/~sikora/ens/graphes/projet2020/sujet.html

Pour l'algo GRASP (article section 6):

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

Documentations Python :

https://www.quennec.fr/trucs-astuces/langages/python/python-comparer-le-contenu-de-deux-%C3%A9l%C3%A9ments

https://docs.python.org/3/library/fileinput.html

https://python.sdv.univ-paris-diderot.fr/07_fichiers/

Signal SIGTERM :

https://www.optil.io/optilion/help/signals