Projet d'algorithme dans les graphes

Challenge PACE 2021

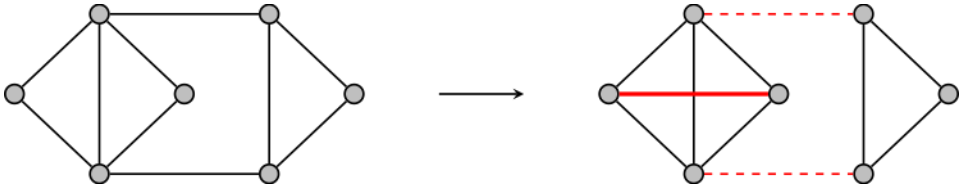
Cluser Editing

Réalisé par : KEBOUCHE Amine

Enseignant : SIKORA Florian

Introduction :

L'objectif de ce projet est de construire un programme qui prend en entré un graphe qui est représenté sous forme de fichier .gr (format imposer par le site) et renvoie un fichier solution qui contient le minimum d'arêtes à modifiées (supprimer/ajouter) dans ce graphe afin d'obtenir un ensemble de clusters (cliques).



Segment rouge : arête ajoutée / Pointillés : arête supprimée

Solution :

Pour résoudre ce problème je me suis inspiré d'un article qui a été proposé par le prof (solver description), et de l'algorithme d'Edmonds pour les couplages connu sous le nom de l'algorithme des fleurs et des pétales.

Mon programme utilise deux algorithmes différents ; le premier que j'ai appelé « Qui sont mes voisins ! » s'exécute sur les graphes denses tandis que le deuxième, inspiré du sparse algorithme , s'exécute sur les graphes creux.

Un graphe *G = < V, E >*, ***(soit |V| = n, et |E|= m*** tel que ***n*** est le nombre de sommets et ***m*** est le nombre d'arêtes**)** est dense si $\frac{2m}{n(n-1)}$ tend vers 1, et il est creux si $\frac{2m}{n(n-1)}$ tend vers 0 autrement dit (nombre d'arête est très petit par rapport aux nombre maximal d'arêtes du graphe G.

Algorithme 1 : « Qui sont mes voisins !»

Cet algorithme s'exécute sur les graphes denses, *[après une analyse des instances publiques et privées en utilisant ce diagramme (Benchmark for Heuristic Track) j'ai constaté que l'algorithme s'exécute sur la majorité des instances].*

- **Principe :**

On part d'une clique qui contient un sommet et on parcourt le reste des sommets du graphe. Pour chaque sommet, on décide de le joindre à une clique déjà existante ou bien d'en créer une nouvelle clique qui va contenir ce dernier. Le choix se fait en comparant le coût d'ajout à une clique et le coût d'en créer une autre.
On note :
On veut classer le sommet ***u*** :

- **i** un sommet qui appartient à une clique
- **j** un sommet du graphe
- ***|C|*** = le nombre de sommets de la clique ***C***
- ***a =*** le nombre d'arêtes ***(u,i)***
- ***b =*** le nombre d'arêtes ***(u,j) = degré de u***
- ***Cost(C,u) = (|C| - a) + (b-a)***
- La fonction ***cost*** désigne le nombre d'arêtes à ajouter et le nombre d'arête à supprimer si on ajoute u à C.
  Pour chaque sommet ***u*** on calcul le ***Cost(C,u)*** pour chaque clique et on choisit la clique qui le ***cost*** le plus petit pour insérer ce sommet ***u.***

Dans mon programme python (algo 1) j'ai choisi de représenté les graphes sous forme de dictionnaire (liste d'adjacence) afin de faciliter la manipulation des cliques en faisant des opérations (intersection, différence). Une fois j'ai l'ensemble des cliques j'appelle ma fonction ***arete_ajout_supp()*** qui compare les arêtes déduites de mon ensemble des cliques avec les arêtes du graphe. (On sait que les sommets appartenant à la même clique sont tous liés deux à deux).

- **<u>Amélioration :</u>**
  Afin d'améliorer mon algorithme et d'utiliser au maximum les 10 minutes accordées, je refais plusieurs fois mon algorithme tout en randomisant le parcours des sommets du graphe et je garde à chaque itération la meilleure solution.

- **<u>Complexité :</u>**
  La complexité de cet algorithme dépend du nombre de cliques que l'on crée à chaque fois, car pour chaque sommet on calcule son cost pour toutes les cliques déjà existantes. Soit ***n*** le nombre de sommets du graphe, alors le nombre de cliques que l'on peut créer est inférieur à **n**, alors la complexité de du programme < O (n*n !).

Algorithme 2 : « Algorithme sur les graphe creux »

Dans cet algorithme, j'ai choisi d'utiliser des listes de listes (liste d'adjacence) afin que je puisse les trier et mettre les sommets en ordre numérique pour pouvoir utiliser la recherche dichotomique lors de la vérification de l'existence ou pas des arêtes.

Je parcours tous les sommets du graphe et j'essaye de trouver des cliques de taille = 3 (triangles) ou de taille = 2, déjà existante, ensuite je fais un parcours randomisé des sommets pour essayer d'augmenter la taille des cliques trouvées (triangles, 2 sommets) tout en modifiant le minimum possible d'arêtes.

Remarque :

- Les graphe de test seront montrés lors de la présentation sur le tableau.
- J'ai utilisé une instruction qui arrête mon programme avant les 10 minutes sur Optil.io pour éviter les TLE (le fichier mainOptilio.py dans le dossier src). Or sur ma machine le fichier main.py répond bien au timeout (10 min).

Bibliographie :

Toutes les sources sont mise en lien cliquable dans ce rapport.