Ginard Guilhem
Zanchetta Enzo

Arbres couvrants de poids minimal d'un graphe, algorithmes voraces : Prim et Kruskal

Ginard Guilhem
Zanchetta Enzo

Table des matières

Ginard Guilhem
Zanchetta Enzo

1. Contexte

La théorie des graphes introduite par Léonard Euler au 18ème siècle est un domaine particulier des mathématiques ayant posé de nombreux questionnements sur des problèmes variés, ayant souvent des solutions algorithmiques simples. Parmi ces problèmes nous pouvons retrouver celui de l'arbre couvrant de poids minimal, étant donné un graphe comme suit :

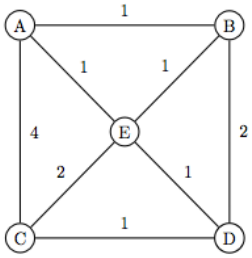


Figure 1 : Exemple de graphe

L'objectif de ce problème est de définir un ensemble d'arrêtes pouvant relier tous les sommets entre eux en tentant de minimiser le poids global (somme) de toutes les arrêtes. Dans notre exemple il vient vite que l'arbre couvrant minimal est le suivant :

$$ArbreCouvrantMin = \{[AE]; [BE]; [ED]; [DC]\}$$

Ce genre de problème peuvent être rencontrés dans diverses domaines concrets, par exemple pour pouvoir créer un réseau électrique ou de transport en desservant toutes les villes tout en minimisant la quantité de matière première nécessaire et donc réduisant les coûts.

Nous allons ici nous intéresser à deux algorithmes dits voraces permettant de trouver ces arbres couvrants de poids minimal, l'algorithme de Prim et de Kruskal.

Un arbre couvrant d'un graphe est un arbre qui relie tous les sommets du graphe.

Un algorithme vorace, quant à lui, est un algorithme qui, à chaque étape, choisira la meilleure solution, afin d'essayer d'obtenir la meilleure solution globale.

2. Algorithme de Prim et Kruskal

Nous allons ici étudier théoriquement les principes des deux algorithmes en alternance pour mieux pouvoir les comparer.

Le principe de l'algorithme de Prim consiste à raisonner en therme de distance entre les points déjà visités et ceux qui ne le sont pas encore. Pour être concis et clair, en choisissant un sommet de départ arbitraire il cherche ensuite l'arrête la plus courte reliant ce premier point à un second, ensuite il cherche l'arrête la plus courte reliant à un troisième point et ainsi de suite. L'algorithme s'arrête lorsque tous les sommets ont été visités.

En ce qui concerne celui de Kruskal on trie les arrêtes par poids croissant dans un premier temps. On ajoute ensuite les deux premières arrêtes de la liste triée à l'arbre couvrant minimal avant de les supprimer de la liste triée. On procède ensuite en ajoutant le chemin le plus court de la liste triée en s'assurant que cela ne forme pas de cycles dans l'arbre. On procède ainsi successivement jusqu'à avoir la liste des chemins triés qui est vide.

En ce qui concerne le meilleur cas dans l'algorithme de Prim cela correspond au cas où le graphe est représenté par des sommets reliés entre eux par une uniques arrêtes et où le sommet choisi arbitrairement correspond à une des deux extrémités du graphe. Dans ce cas l'algorithme se contente à chaque étape de prendre l'unique chemin disponible. Dans le cas de Kruskal tout graphe ne possédant pas de cycle est un meilleur cas, il se contentera alors juste d'ajouter des arrêtes successivement, pour les tests de cyclicité il sera encore plus efficace si l'arbre en construction reste connexe (en un seul morceau dit de manière abusive). Une forme en étoile avec un point central relié à tous les autres sommets conviendrait parfaitement aux propositions précédentes même si elle n'est pas la seule.

Pour le pire cas, que ça soit Kruskal ou Prim ils sont identiques et corresponde à un graphe qui relis chacun de ses sommets à tous les autres. L'explication est simple dans le fait que pour Prim l'algorithme aura un plus gros travail de tri pour définir le chemin le plus court vu qu'il aura plus de possibilités, en ce qui concerne Kruskal c'est le cas où il a le maximum de chance de créer des cycles et donc de passer une itération à éliminer un chemin sans compléter son arbre couvrant minimal.

En ce qui concerne les complexités théoriques des deux algorithmes elles se définissent tel que :

$$\begin{cases} C_{Prim}(n) = o(n^2) \quad avec\ n\ le\ nombre\ de\ sommets \\ C_{Kruskal}(n) = o\big(n \times log(n)\big) \quad avec\ n\ le\ nombre\ d'arrêtes \end{cases}$$

La complexité de l'algorithme de Prim dépend du mode de représentation utilisé pour représenter le graphe. La complexité indiquée ici correspond à une représentation par matrice d'adjacence, celle que nous avons utilisée dans notre algorithme.

3. Choix effectués pour l'implémentation

Pour l'implémentation de ces algorithmes en Python nous avons pris plusieurs choix préliminaires. Le premier concerne la représentation des graphes, nous avons choisi de représenter ces derniers par leur matrice d'adjacence qui est une matrice symétrique donc le coefficient a_{ij} représente la longueur (ou poids) de l'arrête entre les points i et j. Cela nous emmène vers un second choix que nous avons pris pour des raisons pratiques, celui de représenter une absence de chemin par un 0. Ainsi nous avons comme représentation de la matrice d'adjacence du graphe de la *Figure 1* la matrice suivante :

$$A = \begin{pmatrix} 0 & 1 & 4 & 0 & 1 \\ 1 & 0 & 0 & 2 & 1 \\ 4 & 0 & 0 & 1 & 2 \\ 0 & 2 & 1 & 0 & 1 \\ 1 & 1 & 2 & 1 & 0 \end{pmatrix}$$

Pour représenter les matrices, nous avons décidé de dresser des listes de listes. Ainsi la matrice d'adjacence représenté ci-dessus se lit en machine :

```
G=[
    [0,1,4,0,1],
    [1,0,0,2,1],
    [4,0,0,1,2],
    [0,2,1,0,1],
    [1,1,2,1,0]
]
```

Figure 2 : Représentation de la matrice d'adjacence

De plus par soucis de commodité nous avons décidé de représenter les sommets par des numéros, ainsi l'arrête $[AB]$ est représenté par $[1;2]$. Ainsi l'arbre couvrant est représenté par une liste de segments sans doublons (en effet on remarque vite que $[1;2] = [2;1]$).

Pour se servir des algorithmes implémentés il suffit de modifier la matrice d'adjacence dans le fichier *FctBase.py* et ensuite de lancer le fichier correspondant à l'algorithme que l'on veut exécuter.

4. Tests unitaires

Pour les tests unitaires nous avons décidé de nous servir de deux graphes représentatifs des cas pouvant rencontrer les deux algorithmes. Le premier est une simple ligne de 5 sommets (cas favorable de Prim), quant au second il correspond à un pentagone (testant ainsi la réponse dans le cas où l'algorithme peut former un cycle). Pour avoir un résultat attendu et unique nous mettons toutes les arrêtes au même poids sauf celle dans le cas du pentagone entre le premier et dernier sommet, on obtient les matrices suivantes :

$$Test_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Et :

$$Test_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \end{pmatrix}$$

On sait alors que les chemins attendus pour ces deux graphes sont identiques et sont :

$$ArbreCouvrantMin = \big[[1;2];[2;3];[3;4];[4;5]\big]$$

Ces deux tests sont mis en commentaire dans le fichier *FctBase.py* il suffit d'exécuter les programmes en mettant en commentaire l'autre graphe proposé.

Ginard Guilhem
Zanchetta Enzo

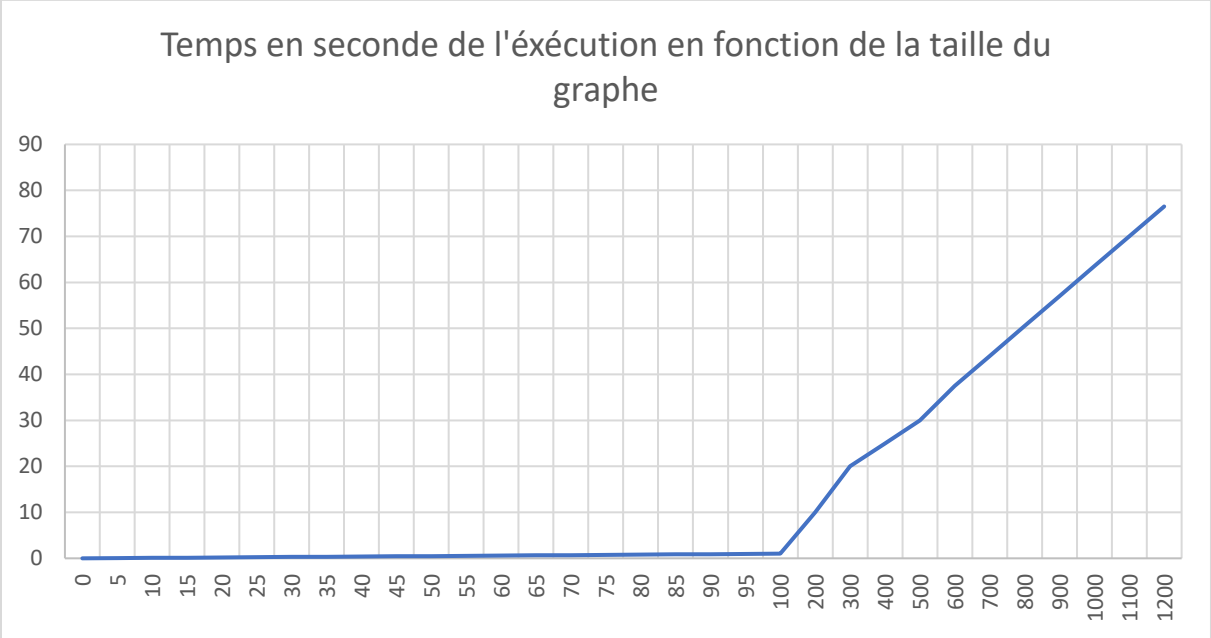
5. Tests de Performances

Pour l'algorithme, l'objectif de ces tests sera de vérifié si le temps d'exécution de l'algorithme en faisant varier la taille du graphe est en adéquation avec une complexité en $o(n^2)$.

Pour cela on lance plusieurs itérations de l'algorithme en faisant varier la taille du graphe, et on observe le temps que mettent les différentes itérations à s'exécuter.

On observe de ces tests que jusqu'à une taille de 100 l'exécution est quasi instantanée. A partir de cette taille, les différentes exécutions commencent a mettre plusieurs secondes jusqu'à une taille de 500 ou là elles comment à mettre plusieurs dizaines de secondes. A partir de 1000, une seule exécution demande plusieurs minutes.

Les données observées semblent correspondre à un algorithme de complexité $o(n^2)$



Nous n'avons malheureusement pas eu le temps de faire les tests de Performances pour l'algorithme de Kruskal mais la méthode reste la même. On fait varier le nombre d'arêtes de graphe ainsi que les dispositions pouvant favoriser les cycles et on observe le temps d'exécution.