Complexité

Coupe minimale d’un graphe

Composition du binome
Mickaël Temim et Mathieu

Document de projet de complexité

Master 1 STL
Sorbonne université
France
4 décembre 2022

Sommaire

Première partie

Exercice 1 : Algorithme de Karger

1 Représentation par matrice d'adjacence

1.1 1.a) Fonction de contraction

Pour ce projet nous avons décidé d'utiliser le langage de programmation "python" pour son coté pratique ainsi que sa facilité à manipuler de tels structures de données. Le fichier "adjacency_matrix_syntaxe.py" contient les fonctions implémentées pour réaliser l'algorithme de Karger à partir d'un graphe représenté par sa matrice d'adjacence. Un tel graphe peut être un graphe classique (chaque arête apparaît une fois maximum) ou un multi graphe (chaque arête peut apparaître plusieurs fois), mais nous ne manipulerons pas de graphes tel que l'arête (i, j) existe si i = j.

Nous avons réalisé une classe "Graphe" représenté par une matrice d'adjacence. Pour cela, nous avons représenté cette matrice par une liste (de taille n), chaque élément de cette liste contient une autre liste (de taille n). Nous avons décidé d'utiliser cette structure de donnée car il n'existe pas de tableau dans ce langage de programmation. Pour la suite, nous considerons qu'on manipuleras un tableau classique de dimension 2.
La methode "contraction" de la classe "Graphe" prend un argument : un tuple 'e' qui représente une arête (a, b) d'un tel graphe. On notera que l'arête (b, a) ainsi que l'arête (a, b) sont équivalentes, néanmoins on partira du principe que a < b (cela est contrôlé par la methode "get_edges_from_id" que nous verrons plus tard). Dans un premier temps nous devons fusionner les sommets a et b. On peut constater que la fusion de ces sommets provequera forcément la suppréssion de l'un de ces deux sommets. Nous choisissons de supprimer le sommet b.
Pour fusionner le sommet a et le sommet b, il faut d'abord copier toutes les arêtes adjacentes à b dans a et supprimer toutes les arêtes (a, b). Nous pouvons ensuite supprimer le sommet b en le supprimant de la matrice.

1.2 1.b) Analyse de la complexité de l'algorithme de contraction sur differentes familles de graphe

Au niveau de la compléxité, nous bouclerons deux fois sur tous les sommets du graphe : une première fois pour copier les arêtes adjacentes à b dans a, et une deuxième fois pour supprimer toutes les arêtes adjacentes à b. Nous avons donc une complexité en O(n). Nous pouvons constater que cette compléxité dépend uniquement du nombre de sommet du graphe et non en fonction du nombre d'arêtes. Le type de graphe utilisé n'influe donc en rien sur cette compléxité contrairement au nombre de sommet de ce dernier.

Nous pouvons néanmoins discuter de l'état du graphe après avoir subit une contraction sur l'une de ses arêtes.
Graphe cyclique :
La contraction d'une arête d'un tel graphe de taille n provoquera un autre graphe cyclique de taille n - 1.
Graphe complet :
La contraction d'une arête (a, b) d'un tel graphe de taille n provoquera un autre graphe complet de taille n - 1 avec une arête suplémentaire entre chaque sommet (a, x) du graphe (on rappelle que a fusionne avec b et que b n'existe plus dans ce graphe).
Graphe bipartie complet :
La contraction d'une arête (a, b) d'un tel graphe de taille n provoquera un autre graphe bipartie complet de taille n - 2 (on enleve les sommets a et b) en plus d'un sommet (le sommet a qui a fusionné avec le sommet b) qui sera lié à tous les autres sommets du graphe.

1.3 1.c) Tirage aléatoire d'une arête

Pour réaliser ce tirage, nous allons associer un identifiant unique à chaque arête numéroté de 0 à m - 1 (m étant le nombre d'arêtes du graphe). Nous n'avons pas besoin d'utiliser un champs supplémentaire "id_edges" pour la simple et bonne raison qu'il est suffisant de réaliser un parcours classique de toutes les arêtes via la matrice d'adjacence (attention, chaque arête apparaît deux fois, il est donc inutile de parcourir toute la matrice).

Pour commencer nous allons tirer un nombre aléatoire compris entre 0 et m - 1 inclus (via la fonction "randint" en python). Ce nombre là sera notre identifiant, nous n'avons plus qu'à parcourir la matrice afin de trouver cette arête (cf la methode "get_edges_from_id" dans le fichier "adjacency_matrix_syntaxe.py").

1.4 1.d) Compléxité de l'algorithme de Karger

Nous avions vu que la compléxité de la contraction dépend uniquement du nombre de sommet du graphe. l'algorithme de karger est un algorithme qui répète plusieurs fois l'algorithme de contraction sur des arêtes choisis au hasard jusqu'à ce que le graphe possède 2 sommets. Nous allons réaliser cette contraction n - 2 fois. Hors, nous avions vu précédemment que la compléxité de la contraction est en O(n). En plus de cette contraction, nous devons explorer la matrice afin de trouver l'arête qui correspond à l'identifiant tiré au hasard précédemment. L'exploration d'une matrice se fait en $O(n^2)$
Nous avons donc : $n * (n^2 + (n - 2)) = O(n^3)$

Cette compléxité est plus coûteuse que celle du cours qui est en $O(n^2)$

2 Représentation par liste d'adjacence

2.1 1.e) Choix de la structure de donnée utilisée

Pour cette deuxième représentation d'un graphe, nous avions décidé d'utiliser une liste d'adjacence. Cette liste sera de taille n et chaque élément de cette dernière stockera une liste qui contiendra tous les sommets adjacents au sommet en question.

Cette représentation se trouve dans le fichier "adjacency_list_syntaxe"

2.2 1.f) Tirage aléatoire d'une arête et compléxité

Ce tirage est presque similaire à celui qu'on a analysé précédemment. Chaque arête apparaît deux fois (tout comme pour la matrice d'adjacence). Nous n'aurons pas d'autre choix que d'explorer la totalité de cette liste d'adjacence contrairement à la matrice d'adjacence où il était possible d'explorer que la moitié de cette dernière. C'est pour cette raison que nous allons multiplier la valeur de l'identifiant par deux. L'élaboration de l'algorithme est exactement la même que pour le tirage précédent. La compléxité de cette algorithme nous oblige donc à explorer la totalité de cette liste, le pire cas serait un graphe complet de taille n où cette compléxité est de $O(n^2)$.
Utiliserunetellestructurededonnepeut−treintrrssantcarcontrairementlamatriced'adjacence, chaquelm

2.3 1.g) Compléxité de l'algorithme de contraction

On suppose qu'on souhaite contracter l'arête (a, b) tel que a < b. On rappelle que le sommet b n'existera plus à la fin de l'algorithme.
Dans un premier temps, on ajoute les arêtes adjacentes à b dans les arêtes adjacentes à a. Pour cela il suffit de fusionner la liste a avec la liste b, cela se fait avec une compléxité qui vaut O(n).
Ensuite, on supprime le sommet b de la liste, cela se fait avec une compléxité qui vaut O(n).
Pour tous sommets x qui sont supérieur à b, x = x - 1. Nous avons pris la décision de faire cette manipulation pour identifier plus facilement les sommets du graphe. Nous devons également modifié tous les sommets b par a dans cette liste, cela implique le faite que nous devons parcourir

la totalité des éléments de cette dernière, la compléxité est donc en $O(n^2)$.

Enfin, on supprime les arêtes (a, a), cela se fait en O(n)

On a donc une compléxité total qui vaut : $n + n + n^2 + n = O(n^2)$

On constate que cette compléxité est supérieur à la contraction d'un graphe représenté par une matrice d'adjacence (O(n)), cela est dù à la réorganisation des sommets (renommage des sommets supérieur à b et modification des sommet b en sommet a). qui a une compléxité en $o(n^2)$.

2.4 1.h) Compléxité de l'algorithme de Karger

La complexité de cette algorithme est éxactement la même que pour l'algorithme de Karger analysé précedemment.
On va itérer n - 2 fois la séléction de l'arête associé à l'identifiant tiré au hasard puis l'algorithme de contraction.

On a donc : $n - 2 * (n^2 + n^2) = O(n^3)$