INFO-F-203 - Rapport

Projet 1

Yahya Bakkali
Matricule : 445166

Maxime Hauwaert
Matricule : 461714

Date : Novembre 2018

Table des matières

1 Introduction générale **2**

2 Sous-arbre de poids maximum **2**
- 2.1 Introduction . . . 2
- 2.2 Choix d'implémentation . . . 2
- 2.3 Algorithme . . . 2
- 2.4 Arbres aléatoires . . . 3

3 Les hypergraphes et hypertrees **3**
- 3.1 Introduction . . . 3
- 3.2 Choix d'implémentation . . . 3
- 3.3 Algorithmes . . . 3
- 3.4 Hypergraphes aléatoires . . . 5

4 Librairies utilisées **5**
- 4.1 Numpy . . . 5
- 4.2 Matplotlib . . . 6
- 4.3 Copy . . . 6

5 Conclusion **6**

1 Introduction générale

Ce projet a pour but de mettre en pratique des concepts sur les graphes vus au cours d'algorithmique 2 pour une meilleure compréhension et maîtrise de ceux-ci.

2 Sous-arbre de poids maximum

2.1 Introduction

Dans ce problème nous manipulons des arbres constitués de nœuds ayant un poids. Le problème consiste à transformer un arbre $T = (V, E)$ en arbre $T' = (V', E')$ de façon à maximiser la fonction

$$w(V') = \sum_{v \in V'} w(v)$$

2.2 Choix d'implémentation

Nous avons décidé de ne pas modifier l'arbre de départ mais de créer une liste qui contiendra le nom de tous les nœuds à désactiver, pour qu'à l'affichage on puisse voir l'arbre de départ avec les nœuds activés (en rouge) ainsi que ceux désactivés (en gris).

2.3 Algorithme

Algorithme 1 maxContribution

```
Require: liste nœuds_à_désactiver
1: poids_total = nœud.poids
2: for chaque enfant du nœud do
3:    if enfant.maxContribution()<= 0 then
4:       Ajouter enfant à nœuds_à_désactiver
5:    else
6:       Ajouter enfant.maxContribution() à poids_total
7:    end if
8: end for
9: return  poids_total
```

La complexité de cet algorithme est de $O(n)$ car il parcourt chaque nœud de l'arbre, $O(n)$ et à chaque nœud on additionne deux valeurs en $O(1)$, on compare deux valeurs en $O(1)$ et on ajoute à

chaque fois[1] le nom du noeud dans une liste donc $O(1)$. Donc la complexité finale est de $O(n)$.

2.4 Arbres aléatoires

Cette génération aléatoire d'arbres a une très bonne distribution. Tous les arbres sont possibles. Il y a de 1 à n nœuds qui composeront l'arbre, 'n' étant 15 dans ce projet. Chaque nœud choisira tout simplement de qui il veut être l'enfant parmi les nœuds déja placés.

3 Les hypergraphes et hypertrees

3.1 Introduction

3.2 Choix d'implémentation

3.3 Algorithmes

Algorithme 2 find_cliques

Require: R : {nœuds d'une clique maximale}, P : {nœuds possibles dans une clique maximale}, X : {nœuds exclus}

```
1:  if P et X sont vides then
2:    if la clique R est de taille >= 2 then
3:      Ajouter R a la liste des cliques
4:    end if
5:  else
6:    pivot = élement aléatoire de l'ensemble P ∪ X
7:    for chaque sommet S dans l'ensemble P \ {sommets liés au pivot} do
8:      newP = P ∩ {sommets liés à S}
9:      newR = R ∪ {S}
10:     newX = X ∩ {sommets liés à S}
11:     find_cliques(newP,newR,newX)
12:     P = P \ {S}
13:     X = X ∪ {S}
14:   end for
15: end if
```

Tout graphe à n sommets a au maximum 3n/3 cliques maximales, et le temps d'exécution le plus défavorable de l'algorithme de Bron-Kerbosch (avec une stratégie pivot qui minimise le nombre d'appels

[1] Au pire des cas

récursifs effectués à chaque étape) est $O(3n/3)$, correspondant à cette limite.

Algorithme 3 is_chordal

```
1: unnumbered = ensemble des sommets du graphe
2: s = sommet choisi aléatoirement dans unnumbered
3: unnumbered = unnumbered \ {s}
4: numbered = {s}
5: while unnumbered ! = {∅} do
6:    Vertex = le sommet de unumbered qui a le plus de connections aux sommets de numbered
7:    unnumbered = unnumbered - Vertex
8:    numbered = numbered + Vertex
9:    clique_wanna_be = {sommets liés à Vertex} ∩ numbered
10:   subGraph = Un sous-graphe induit des sommets appartenant à clique_wanna_be
11:   if le subGraph n'est pas complet then
12:      return False
13:   end if
14: end while
15: return True
```

Au debut, nous créons I $O(N)$, choisissons un sommet arbitraire I $O(N)$, enlevons ce sommet du I $O(1)$, créons II contenant ce sommet $O(1)$. Ensuite, tant que I n'est pas vide $O(N)$ nous chercherons un sommet "Vertex" dans I qui a plus de connexions aux sommets dans II $O(S * N * W) = O(N^2)$ parce qu'on toujours la cardinalite de l'ensemble S+W egale a N, après nous supprimons ce sommet du I en l'ajoutant au II $O(1)$ puis créons un ensemble contiendra l'intersection entre II et III $O(\min(W, len(III)))$ et le sous-graphe induit a partir de cet ensemble $O(N * \min(W, len(III)) * \min(V, len(III)))$. Ce qui fait en final une complexite de $O(N * \max(N^2, \min(W, len(III)), N * \min(W, len(III)) * \min(N, len(III))))$ $= O(N^2 * \min(W, len(III)) * \min(N, len(III)))$

Statique :
III : ensemble des sommets liés au sommet "Vertex"
N : nombre du sommets du graphe

Dynamique :
I : ensemble des sommets du graphe "unnumbered"
II: ensemble des sommets "numbered"

S : nombre du sommets dans "unnumbered"
W : nombre du sommets dans "numbered"

Algorithme 4 Algorithm_X

Require: Matrice

Faire une copie de la matrice et exécuter l'algorithme sur cette matrice
Choisir la colonne C contenant un minimum de 1
L = L'ensemble des lignes tel que $Matrice_{l,c} = 1$, $\forall l \in L$
for chaque ligne l de L **do**
    columnslist = []
    rowslist = []
    Ajouter la ligne à la solution partielle
    for chaque colonne j de la matrice **do**
        if $Matrice_{l,j} = 1$ **then**
            for chaque ligne i de la matrice **do**
                if $Matrice_{i,j} = 1$ **then**
                    Ajouter la ligne i à rowlist
                end if
            end for
            Ajouter la colonne j de la matrice à columnslist
        end if
    end for
    Supprimer les lignes et les colonnes de la matrice présentes dans rowslist et columnslist
    if la matrice n'est pas vide **then**
        if toutes les colonnes de la matrice ont au moins un 1 **then**
            Répéter cet algorithme de façon récursive sur la matrice réduite
        end if
    else
        Ajouter la solution à l'ensemble des solutions
    end if
    Supprimer la ligne de la solution partielle
    Réutiliser la matrice de départ
end for

3.4 Hypergraphes aléatoires

4 Librairies utilisées

4.1 Numpy

C'est une librairie très utile dans ce projet pour l'utilisation d'opérations mathématiques telles que les fonctions sinus/cosinus, etc ainsi que dans la manipulation de l'aléatoire.

4.2 Matplotlib

C'est une librairie assez utile dans ce projet pour l'affichage d'objets mathématiques en 2D tels que des cercles, des lignes, etc.

4.3 Copy

C'est une librairie contenant la fonction "deepcopy" permettant de copier l'intégralité d'un objet sans qu'il n'y ait de liens entre l'ancien et le nouvel objet.

5 Conclusion

En plus de la simple mise en pratique de certains concepts sur les graphes, ce projet nous a permis de développer nos compétences de travail en groupe.