|  |
| --- |
| INFO-F203 Algorithmique 2 |
| Projet 1 |
|  |



# Introduction

Ce document est un rapport qui présente notre projet dans différents points de vue.

Nous allons présenter dans un premier temps la structure de donnée de notre programme ainsi que les choix qui ont menés à l’utiliser.

Dans un second temps nous étudierons l’efficacité de nos algorithmes et terminerons par leur complexité.

# Structure de données et choix d’implémentation

Notre programme utilise des libraires externes à Python 3. Nous utilisons une librairie externe nommée *Networkx* sous sa version *2.2 datant de Septembre 2018*. Celle-ci utilisera la libraire *Matplotlib* de façon interne. *Networkx* est une librairie permettant de créer une structure de donnée modélisant des graphes. Elle nous sera donc très utile pour représenter notre structure graphiquement.

Hormis cette librairie, tous nous algorithmes sont basés sur des structures de données créées spécialement pour le programme. Nous les présentons ici.

Nos structures se présentent majoritairement sous forme de classes.

## Partie 1 : max\_subtree

Cette première partie nous demandais de créer une fonction *max\_subtree()* prenant en paramètre un arbre enraciné et le renvoyant maximisé en fonction du poids des nœuds.

L’arbre donné en paramètre est une de nos structures de données sous forme de classe. Il appartient donc à une classe nommée *WeightedTree*.

Cette classe est simple. Ses attributs et méthodes mentionnés sont propres à la structure d’un arbre et demandés presque explicitement dans l’énoncé.

### Représentation de l’arbre

Parmi les attributs, nous retrouvons :

Le nom du sommet, la référence vers le père du sommet, son poids, la liste des successeurs du sommet, le nombre de successeurs et son potentiel.

Ce dernier attribut sera ici un élément clé pour maximiser notre arbre. Nous expliquerons son utilité quand nous viendrons à parler du fonctionnement de l’algorithme de maximisation.

En ce qui concerne les méthodes de notre classe, elles sont au nombre de 12.

7 sont des *getters* et 2 des *setters*. Les *getters* renvoient simplement l’état d’un attribut. Les *setters* modifient l’attribut *potentiel* et *liste des successeurs* respectivement. Quant aux 3 autres méthodes, elles modifient la liste des successeurs en ajoutant, supprimant ou modifiant du contenu.

Nous avons choisi de développer notre propre structure ainsi afin d’avoir un contrôle lors d’erreurs pendant la phase de programmation. Celle-ci est claire et permet donc de corriger l’erreur rapidement sans modifier complètement la structure. Ce choix nous parait le plus simple et logique. *Networkx* propose une structure d’arbre mais celle-ci ne convenait pas du tout lors de la maximisation. Nous transformons notre structure d’arbre en celle-ci lors de la représentation graphique.

### Maximisation de l’arbre

Une fois notre arbre créé, il est envoyé à la fonction *max\_subtree()*.

L’idée de l’algorithme est la suivante : Effectuer un parcours en profondeur – *Depthfirst* – afin de de traiter les feuilles en premier.

Expliquons ici l’intérêt de l’attribut *Potentiel*. Celui-ci représente la jauge permettant de supprimer ou non le nœud en question. Il est initialisé à la valeur de son propre poids par défaut. Chaque nœud fils fait monter le potentiel du nœud père si son potentiel est positif. Si, à la fin du parcours de tous les fils, le potentiel du père n’est pas monté au-dessus de zéro, nous effectuons une suppression du père et supprimant ainsi au passage tous le sous-arbre. Si le potentiel est au-dessus de zéro, nous gardons le sous-arbre et faisons monter le potentiel du nœud ancêtre.

Voici un pseudo-code de la fonction max\_subtree()

1 : for *sommet* in *liste\_des\_fils* : # range(0) pour une feuille donc <<stop>> et suite

2 : max\_subtree(sommet) # Appel récursif

3 : si sommet != racine :

4 : si *sommet*=feuille :

5 : si *poids* négatif :

6 : supprimer la feuille

7 : sinon faire monter le potentiel du père

8 : sinon :

9 : si *potentiel* <= 0 :

10 : faire monter le potentiel du père

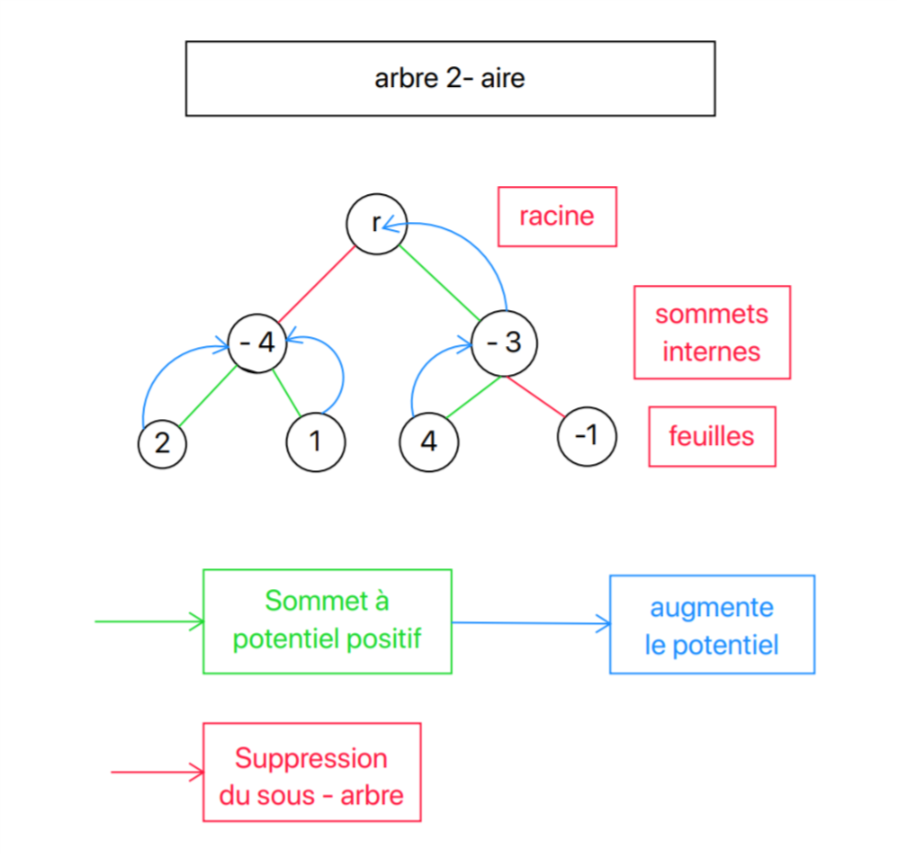
11 : sinon supprimer le nœud # donc tout le sous-arbre

12 : si sommet est la racine :

13 : si *potentiel* nul ou négatif : Pas de sous arbre et supprimer ses fils restants

Nous adoptons une approche récursive pour se fixer un cas de base – ici c’est lorsque nous atteignons une feuille – lequel nous permet d’être surs que le potentiel de chaque fils de chaque niveau inférieur est positif.

Voici un schéma représentant une situation et donnant une idée du fonctionnement de la fonction.



A la fin de la fonction, celle-ci renvoie l’arbre modifié et maximisé. Le cas où il n’existe pas de sous-arbre de poids maximum est également traité car tous les sous-arbres de la racine de potentiel inférieur à un sont supprimés. Ainsi à la fin de la fonction, si le potentiel de la racine n’est pas monté au-dessus de zéro, la fonction renvoie donc l’ensemble vide.

### Complexité

Intéressons-nous à la complexité de notre fonction *max\_subtree()*. Nous savons que celle-ci effectue récursivement un parcours en profondeur de l’arbre, parcourant ainsi une et une seule fois chacun de ses sommets. Nous pouvons alors dire que la complexité d’un tel parcours se fait en un temps linéaire *O(n)*. A cela s’ajoute les vérifications et instructions de modification supplémentaires. La fonction parcourt bel et bien chaque sommet vu que la suppression éventuelle ne commence qu’après avoir traité tout le sous arbre correspondant (voir pseudo code). L’opération de suppression d’un nœud se fait dans le pire des cas en *O(n)* car nous pourrions supprimer le dernier élément de la liste et ainsi parcourir toute la liste. Pour la modification du potentiel du père, cela se fait en temps constant *O(1)*.

## Partie 2 : Hypergraphes et hypertrees

La deuxième partie de l’énoncé nous demandais de représenter le graphe *Dual* d’un hypergraphe donné et créé aléatoirement ainsi que de vérifier si l’hypergraphe est un *hypertree*.

### Représentation de l’hypergraphe

La représentation d’un hypergraphe se fera également sous forme d’une classe*. Bibaprtite\_graph*.

Un graphe biparti donc. Nous avons fait le choix de représenter directement un hypergraphe sous forme de graphe biparti car cela nous facilite la création du graphe dual. En effet, il nous suffira d’intervertir les sommets et hyper-arêtes.

Les méthodes implémentées ne sont que des getters des attributs.

La classe *Bibaprtite\_graph* se compose d’une liste de sommets et d’une liste d’hyper-arêtes. Les sommets non inclus dans une hyper-arête son dans une liste à part. Une liste de tuples hyper-arête - sommet est également présente en vue de la création d’un dictionnaire servant à la création du graphe primal. De plus, nous créons avec Networkx le graphe associé. Son utilité sera expliquée plus loin.

Notons au passage qu’un hypergraphe Dual n’est rien d’autre qu’un hypergraphe simple.

Une classe représentant un graphe primal est également créé car il fait l’objet de création de méthodes spécifiques notamment une méthode isChordal() expliqué au point suivant.

### Graphe α-acyclique

Une fois notre graphe dual et son graphe primal créé, nous pouvons passer à la vérification de son acyclicité. Notons que le graphe dual avec une représentation en graphe biparti est le même hormis l’échange entre hyper-arêtes et sommets mais le graphe primal est bien différent.

Pour vérifier qu’un graphe primal est cordal nous utilisons la méthode fournie par Networkx. La méthode isChordal(). Sa complexité sera expliquée dans sa section. Cette méthode renvoie True ou False selon que le graphe l’est ou non.

*« En théorie des graphes, on dit qu'un graphe est cordal si chacun de ses cycles de quatre sommets ou plus possède une corde, c'est-à-dire une arête reliant deux sommets non-adjacents du cycle. »[[1]](#footnote-1).* L’algorithme de Networkx utilisé est consultable à cette adresse[[2]](#footnote-2).

L’algorithme fait une recherche de cliques pour chaque sommet. Il va donc trouver toutes les cliques de taille 4 ou plus qui contiennent le sommet courant contiennent une arête reliant deux sommets non adjacents de cette clique (par définition).

En ce qui concerne la vérification des cliques, la méthode encore fournie par Networkx est utilisée. Cette méthode nous fournit un itérateur sur les cliques lesquelles seront comparées aux hyper-arêtes de l’hypergraphe grâce à la fonction *max\_cliques().*

TO DO EXPLIQUER L’ALGORITHME get\_max\_cliques() et complexité

Max\_cliques() ne fait que garder les cliques du graphe primal de taille 2 ou plus et les compare aux sommets inclus dans une hyper-arête. Si tel est le cas la variable *res* initialement créée reste à True.

### Complexité

is\_chordal() : La complexité de l’algorithme utilisé par Networkx se fait au pire des cas en *O(m+n)* où *m* représente le nombre de nœuds et *n* le nombre d’arêtes dans le graphe. Une preuve de ce résultat peut se trouver à cette adresse[[3]](#footnote-3).

Table des matières

[1. Introduction 1](#_Toc529954089)

[2. Structure de données et choix d’implémentation 1](#_Toc529954090)

[2.1. Partie 1 : max\_subtree 1](#_Toc529954091)

[2.1.1. Représentation de l’arbre 2](#_Toc529954092)

[2.1.2. Maximisation de l’arbre 2](#_Toc529954093)

[2.1.3. Complexité 4](#_Toc529954094)

[2.2. Partie 2 : Hypergraphes et hypertrees 4](#_Toc529954095)

[2.2.1. Représentation de l’hypergraphe 4](#_Toc529954096)

[2.2.2. Graphe α-acyclique 4](#_Toc529954097)

[2.2.3. Complexité 5](#_Toc529954098)

1. ***Graphe cordal :* définition.** <http://fracademic.com/dic.nsf/frwiki/729771> consulté le 13-11-2018. [↑](#footnote-ref-1)
2. **Networkx : is chordal :** <https://github.com/networkx/networkx/blob/master/networkx/algorithms/chordal.py#L284> consulté le 13-11-2018. [↑](#footnote-ref-2)
3. **Chordal Graphs: Their Testing and Their Role** , J-F Huard : <https://pdfs.semanticscholar.org/282c/729fe589f328bbe8937714970d15dbb7f18a.pdf> pp.7 consulté le 13-11-2018. [↑](#footnote-ref-3)