Laamiri Achraf, El Achouchi Iliass

Rapport : Projet d'Algorithmique 2

Université Libre de Bruxelles
Science informatique
info-F-203

Année accadémique 2018-2019

1 Arbre

1.1 Implémentations

1.1.1 Entrée et Sortie

Tout ce qui est en rapport avec les sorties de ce projet se fera via les librairies `networkx` et `matplotlib`. Elles offrent un choix assez large quant à l'affichage et la gestion de graphes, ce qui facilite grandement la partie visualisation du projet.
Par contre les entrées pour cette partie du projet se feront via l'algorithme de génération aléatoire d'arbre `random_tree`.

1.1.2 Affichage

L'exécution de cette partie du projet ouvre une seule et unique fenêtre grâce à la méthode `show` de `matplotlib.pyplot`. Elle affiche sur la partie supérieure de la fenêtre l'arbre généré, et juste en dessous l'arbre réduit, conformément à l'énoncé du projet. Cette vue permet de comparer assez facilement l'arbre généré et l'arbre réduit.

Les nœuds de l'arbre sont nommés, ils sont enracinés en « r » et ont chacun une valeur comprise entre -5 et 5 afin de rester raisonnable dans la génération d'arbre aléatoire.

Enfin, d'un point de vue purement esthétique, un algorithme qui permet de "balancer" l'arbre a été implémenté, mais il n'est malheureusement pas de nous [1].

L'objet Tree est un objet récursif, ou chaque fils d'un arbre est lui-même un arbre. L'algorithme demandé, `max_subtree`, a été implémenté sous forme de méthode de la classe `Tree`. Il permet d'agir directement sur un objet `Tree`.

L'algorithme fonctionne comme cela : Pour chaque fils d'un nœud, il observe si la somme complète de l'arbre (calculée avec la méthode `get_subSum` détaillée plus bas) est inférieure ou égale à 0. Dans ce cas, il supprime le nœud et ses enfants s'il y en a. Le résultat sera donc une somme ou l'arbre sera positive ou rien du tout. Dans le cas ou il n'y a pas de sous-arbre positif de poids positif, il n'y a que l'arbre principal qui est affiché. Le paramètre G de la méthode permet simplement de gérer l'affichage de `networkx`, l'algorithme supprime le nœud du graphe `networkx` au même moment où il supprime le nœud qui fait partie de l'arbre.

La méthode `get_subSum` est très importante. En effet, elle permet de faire la condition de suppression d'un nœud dans la méthode `max_subtree`. Le parcours est similaire à `max_subtree`, on parcourt chaque fils d'un nœud et on ajoute à

1. https ://stackoverflow.com/questions/29586520/can-one-get-hierarchical-graphs-from-networkx-with-python-3

une certaine somme la valeur des nœuds enfants. On récupère ainsi la sous somme totale d'un sous-arbre.

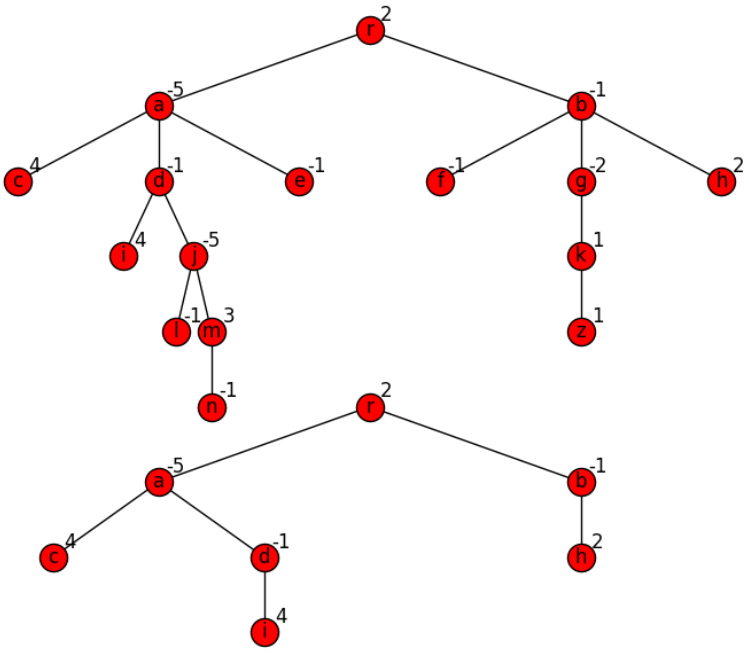


FIGURE 1 – Arbre de l'énoncé et son homologue maximisé

1.1.3 Fonctions et méthodes

1.2 Complexité

1.2.1 `get_subSum`

```python
def get_subSum(self, res=0):
    """
    Algorithme calculant la somme des poids totaux d'un Tree, en effectuant un parcours en profondeur
    Complexité : O(m)
    :param res: somme
    :return: somme des poids totaux d'un Tree
    """
    if self.getChildren() == []: #O(1)
        res += self.getVal() #O(1)
    else:
        res_t = 0  # Somme temporaire afin de l'ajouter totalement à res plus tard #O(1)
        for c in self.getChildren(): #O(m)
            res_t += c.get_subSum(res) #O(m = nombre de noeuds de l'arbre)
        res += self.getVal() + res_t #O(1)
    return res #O(1)
```

FIGURE 2 – Fonction `get_subSum`.

Pour cette fonction, la ligne 8 et 9 sont en $O(1)$ car on appelle un accesseur et on effectue une somme. Même chose pour la ligne 14. En ce qui concerne la boucle à la ligne 12, la ligne 13 est en $O(m)$ ou m est le nombre de nœuds dans l'arbre car on va l'appliquer sur tous les nœuds de l'arbre. Enfin, la boucle est en $O(m)$ où m est le nombre de nœud de l'arbre complet.

1.2.2 `max_subtree`

```
1   def max_subtree(self, G):
2       """
3       Algorithme principal demandé. Supprime des noeuds s'il considère qu'ils ne contribuent au score max du Tree
4       Complexité O(n*m)
5       :param G: graphe networkx utilisé pour l'affichage
6       :return: Rien
7       """
8       i = 0 #O(1)
9       while i < len(self.getChildren()): #O(n*m)
10          self.getChildren()[i].max_subtree(G) #O(n*m)
11          if self.getChildren()[i].get_subSum() <= 0:#O(m)
12              G.remove_node(self.getChildren()[i].getRoot())
13              del self.getChildren()[i] #O(n)
14
15              i -= 1 #O(1)
16          i += 1 #O(1)
```

FIGURE 3 – Fonction `max_subtree`.

Cette fonction récursive a des instructions de $O(1)$ à la ligne 8, 15 et 16. À la ligne 13, nous avons cependant une complexité en $O(n)$ car la suppression d'un valeur dans une liste demande de réécrire toute la liste sans l'élément supprimé. Pour l'instruction `if` à la ligne 11, la complexité maximale est en $O(m)$ car c'est la complexité de la fonction `get_subSum` calculée plus haut. En ce qui concerne celle de l'instruction de la ligne 10, nous avons déduit que le nombre d'appel de la fonction récursive est en $O(n*m)$ où n est le nombre de fils.

Nous pouvons enfin dire que la complexité maximale finale de la fonction est de $O(n * m)$. Néanmoins, nous pensons que la complexité est légèrement inférieure à cette valeur, car comme le montre la ligne 13, le nombre de fils varie, et donc par la même occasion le nombre de nœuds. Il ne faut pas non plus oublier que dans notre cas, les valeurs m et n sont assez petite. Elles varient entre 0 et 15, et 0 et 3 respectivement.

2 Hypergraphe

2.1 Implémentations

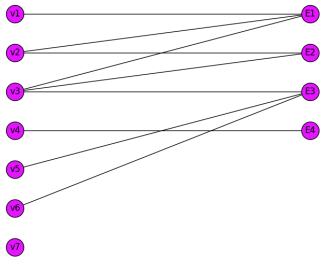
2.1.1 Entrée et Sortie

En ce qui concerne l'entrée, nous avons opté pour une matrice d'incidence car il était plus simple de schématiser les arrêtes par une valeur booléenne. De plus, il était facile, via la librairie `Numpy` et sa méthode!!!random!!!, de générer des matrices aléatoires, ainsi que pour l'affichage qui sera décrit ci-dessous.

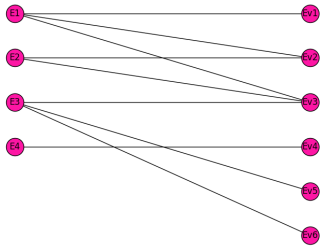
2.1.2 Affichage

Nous avons décidé d'afficher le graphe d'incidence, via la matrice d'incidence, le graphe dual, via une simple transposée de la matrice d'incidence et enfin le graphe primal du graphe dual. De cette façon, nous pouvons afficher plus facilement toutes les informations du graphe et de les comparer.

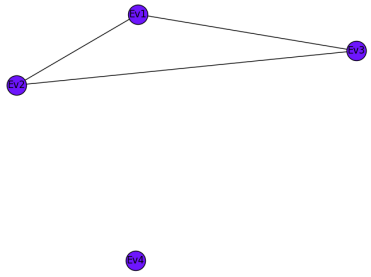
Le terminal indiquera finalement si l'hypergraphe est un hypertree comme demandé. La matrice d'incidence est également affichée.



(a) Graphe d'incidence.



(b) Graphe dual.



(c) Graphe primal.

FIGURE 4 – Affichage des différents graphes.

2.1.3 Fonctions et méthodes

En ce qui concerne le fonctionnement des méthodes et fonctions de notre implémentation, nous avons essayé de respecter un maximum la mécanique d'encapsulation, via l'utilisation de plusieurs accesseurs et mutateurs dans notre classe `Hypergraph`.

Un objet de cette classe prend comme paramètre une matrice d'incidence, et via cette-dernière, va créer sa transposée. Avec ces 2 matrices, les graphes dual,

primal et d'incidence seront créés.

Pour pouvoir vérifier si un graphe est un hypertree ou non, nous devons tout d'abord vérifier s'il est cordal ou non. C'est ce que fait exactement la méthode `is_chordal` de `networkx`. Elle prend en paramètre un graphe et retourne un booléen si oui ou non le graphe est cordal.

Si cette condition est vérifiée, on va analyser chaque clique maximale donnée par la méthode networkx `find_cliques` dans la méthode `checkClique`. Cette dernière va avant tout créer un dictionnaire des hyperarêtes avec leurs sommets respectifs, pour ensuite comparer chaque clique maximale avec les sommets des hyperarrêtes. Si toutes les cliques sont présentes dans les hyperarêtes, alors la méthode renvoie `True`, et `False` sinon.

2.2 Complexité

Table des matières