Rapport Projet 1
INFO-F-203

Antoine Lemahieu, 457582
Tristan Philips, 425461

14 novembre 2018

Sommaire

Partie 1 : Arbres

Brève introduction

Dans la première partie de ce projet, il a été demandé de générer un arbre, où chaque sommet a un certain poids, et d'en trouver le sous arbre de poids maximum. Autrement dit, il faut amputer l'arbre initial des sous-arbres qui ont tendance à faire baisser son poids et de n'en garder que les parties qui contribue à le rendre plus élevé.

Librairies et fonctions importées

Par soucis de simplicité, nous avons décidé d'utiliser les librairies Networkx et Matplotlib. L'une servant servant à créer l'arbre, composés de ses différents noeuds, l'autre à l'afficher. La librairie random a bien évidemment été utilisée afin de générer des arbres aléatoires de manière à pouvoir tester une multitude de cas différents.

Pour faciliter la présentation de l'arbre, nous avons utilisé une fonction trouvée sur internet [1], qui a ensuite été modifiée partiellement par nos soins. Celle-ci renvoie un dictionnaire où chaque noeud à une position x et y associée.

Choix d'implémentation faits et fonctionnement du programme

Afin d'éliminer au fur et à mesure les feuilles et sous arbres contribuant à faire baisser le poids total de l'arbre, il nous a semblé évident d'effectuer un parcourt en post ordre. Un tel chemin consiste à parcourir les fils de gauche à droite, puis le père. Il est ainsi aisé de visiter toutes les feuilles, d'éliminer celles ayant un poids négatif et d'ensuite aditionner le poids des feuilles restantes (qui sont positifs donc) au poids du père. Si l'ensemble est positif, on laisse tel quel et on le considère dans la suite du code comme étant une simple feuille dont le poids correspond à la somme du poids ses feuilles et de lui même. Si l'ensemble est négatif, on élimine le noeud ainsi que tous ses fils. On fait ceci de manière récursive sur l'ensemble des l'arbre.

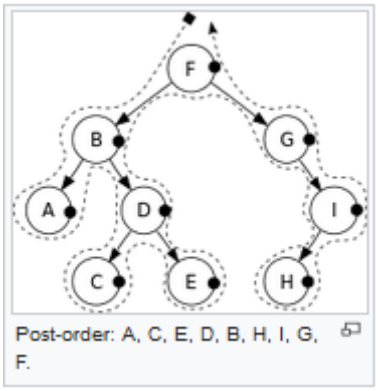

Figure 1 – Exemple d'un parcourt en post ordre [2]

En vue de créer un arbre non binaire, nous avons implémenté une fonction gen_sommet, qui génère aléatoirement entre 10 et 15 sommets de poids allant de -10 à 10. Pour ce faire, à chaque itération de la

boucle principale, un nouveau sommet se rattache alétoirement à un sommet déjà existant. Le deuxième à être créé se liera forcément avec le premier, mais le troisième sera, par exemple, soit le fils du second, soit le fils de la racine (et frère du second). Et ainsi de suite. Nous avons pas mis de condition à cette fonction, il se peut donc que des cas dégénérés (bien que rares) surviennent, telle qu'une racine avec 10 fils, une profondeur d'arbre de 10, ect...

De manière à optimiser l'affichage de l'arbre, nous avons décidé de ne pas afficher les noms des noeuds dans l'aperçu, étant donné que cela représente peu d'intéret mais d'uniquement montrer les poids des sommets. Pour ce faire, nous avons utilisé la méthode get_node_attribute (et par extension le fait que l'on puisse donner des attributs à chaque noeud) de la libraire Networkx.

Complexité des méthodes utilisées et max_subtree

La complexité de la fonction générale max_subtree est de l'ordre de O(n * profondeur de l'arbre), sachant que n symboliserait le nombre de sommets que contient l'arbre. La complexité de toutes les sous-fonctions à été indiquée directement dans le code.

Voici le code source et la complexité des méthodes de la librairie Networkx que nous avons utilisées.

```
if attr_dict is None:
    attr_dict = attr
else:
    try:
        attr_dict.update(attr)
    except AttributeError:
        raise NetworkXError(
            "The attr_dict argument must be a dictionary.")
if n not in self.node:
    self.adj[n] = self.adjlist_dict_factory()
    self.node[n] = attr_dict
else:  # update attr even if node already exists
    self.node[n].update(attr_dict)
```

FIGURE 2 – Code source de la méthode add_node ; O(n)

```
adj = self.adj
try:
    nbrs = list(adj[n].keys())  # keys handles self-loops (allow mutation later)
    del self.node[n]
except KeyError:  # NetworkXError if n not in self
    raise NetworkXError("The node %s is not in the graph." % (n,))
for u in nbrs:
    del adj[u][n]   # remove all edges n-u in graph
del adj[n]          # now remove node
```

FIGURE 3 – Code source de la méthode remove_node ; O(n)

```
        if attr_dict is None:
            attr_dict = attr
        else:
            try:
                attr_dict.update(attr)
            except AttributeError:
                raise NetworkXError(
                    "The attr_dict argument must be a dictionary.")
        # add nodes
        if u not in self.node:
            self.adj[u] = self.adjlist_dict_factory()
            self.node[u] = {}
        if v not in self.node:
            self.adj[v] = self.adjlist_dict_factory()
            self.node[v] = {}
        # add the edge
        datadict = self.adj[u].get(v, self.edge_attr_dict_factory())
        datadict.update(attr_dict)
        self.adj[u][v] = datadict
        self.adj[v][u] = datadict
```

FIGURE 4 – Code source de la méthode add_edge ; O(n)

```
try:
    del self.adj[u][v]
    if u != v:  # self-loop needs only one entry removed
        del self.adj[v][u]
except KeyError:
    raise NetworkXError("The edge %s-%s is not in the graph" % (u, v))
```

FIGURE 5 – Code source de la méthode remove_edge ; O(1)

Partie 2 : Hypergraphes

Brève introduction

Dans cette seconde partie, le but était de générer un hypergraphe et de vérifier s'il s'agissait ou non d'un hypertree. Pour qu'un hypergraphe soit un hypertree, son hypergraphe dual doit être alpha-acyclique. Il faut donc que le graphe primal de l'hypergraphe soit cordal et que toute clique maximale soit une hyper-arête dans l'hypergraphe.
Nous expliquerons plus bas dans la section «Vérification de l'hypertree » comment nous avons procédé pour vérifier toutes ces conditions.

Choix des librairies utilisées

Tout comme pour la première partie de ce projet, nous avons utilisé les librairies networkx et Matplotlib. Nous n'en rejustieront donc ni l'utilité ni la complexité étant donné que tout cela a déjà été fait précédemment dans ce rapport. De même, la fonction «hierarchy position [1]» a de nouveau été utilisée afin d'afficher le graph dual (voir plus bas).

Génération de l'hypergraphe

A l'inverse de la première partie du projet où les arbres étaient des objets d'une classe arbre, notre hypergraphe est seulement une liste de liste. Les sommets d'indice i dans la liste principale sont une sous-liste contenant toutes les hypers-arêtes les incluants. Afin de créer une telle structure, nous avons au préalable décidé de générer entre 5 et 15 sommets différents et entre 3 et 5 hyper-arêtes. Chaque hyper-arête a une probabilité de 50% d'être liée à chaque sommet.

Vérification de l'hypertree

Le principe d'hypertree ayant été défini, il nous faut donc des fonctions créant l'hypergraphe dual ainsi que le graphe primal.

La cordalité d'un graphe peut être vérifiée grâce au principe d'ordonnancement à élimination parfaite [3]. Pour cela, nous allons utiliser la recherche lexicographique en largeur [4] qui «prépare» la structure. La recherche lexicographique fonctionne comme suit : nous créons tout d'abord une liste contenant une autre liste dans laquelle tous les sommets sont présents, c'est notre premier ensemble. À chaque itération, le premier sommet du premier ensemble est choisi. Ensuite nous cherchons chaque sommet voisin avec le sommet choisi et ceux-ci sont placés dans un nouvel ensemble précédant le premier. L'algorithme poursuit ainsi jusqu'à ce que l'entièreté des sommets soient traités.
Par la suite, il faut vérifier que l'ordonnancement obtenu soit d'élimination parfaite. Pour cela, nous passons en revue tous les sommets dans la liste précemment obtenue. Si pour chacun, l'ensemble des voisins qui les précéde forment une clique, alors l'hypergraphe est cordal. Sinon, l'hypergraphe n'est pas cordal et de facto n'est pas non plus un hypertree.

D'autre part, l'algorithme de Bron-Kerbosch [5] permet de prouver que pour chaque clique maximale du graphe, il y a une hyper-arête dans l'hypergraphe qui y correspond, et donc ainsi d'affirmer que l'hypergraphe est un hypertree (à condition que la cordalité soit vérifiée). Grâce à un simple backtracking, l'algorithme va lister l'ensemble des cliques maximales du graphe et va vérifier au fur et à mesure qu'il y a bien une hyper-arête correspondante (en l'occurence, l'hypergraphe dual nous donne directement

l'information dont on a besoin). Dès qu'une clique maximale est trouvée et ne concorde pas, l'algorithme s'arrête et renvoie un résultat erroné. Sinon, il continue jusqu'à avoir comparé toutes les cliques.

Si ces deux conditions sont vérifiées, l'hypergraphe est en effet bien un hypertree.

Complexité des différentes fonctions et test_hypertree

La complexité de la fonction principale test_hypertree est de O(n^3*log(n)). Par soucis de facilité, la complexité des différentes techniques algorithmiques a été indiquée et détaillée directment dans le code.

Représentation de l'hypergraphe dual

Il était demandé d'afficher à l'écran le graph dual de la façon que l'on jugeait la plus simple. Pour ce faire, nous avons décidé de représenter notre graphique en tant que plusieurs arbres de hauteur 1. La racine de chaque arbre correspond à une hyper arête et tous ses fils sont les sommets qu'elle relie. L'affichage est certes, bien moins élégant que celui proposé dans l'énoncé, mais nous trouvions que celui-ci était tout aussi clair et donnait autant d'informations.

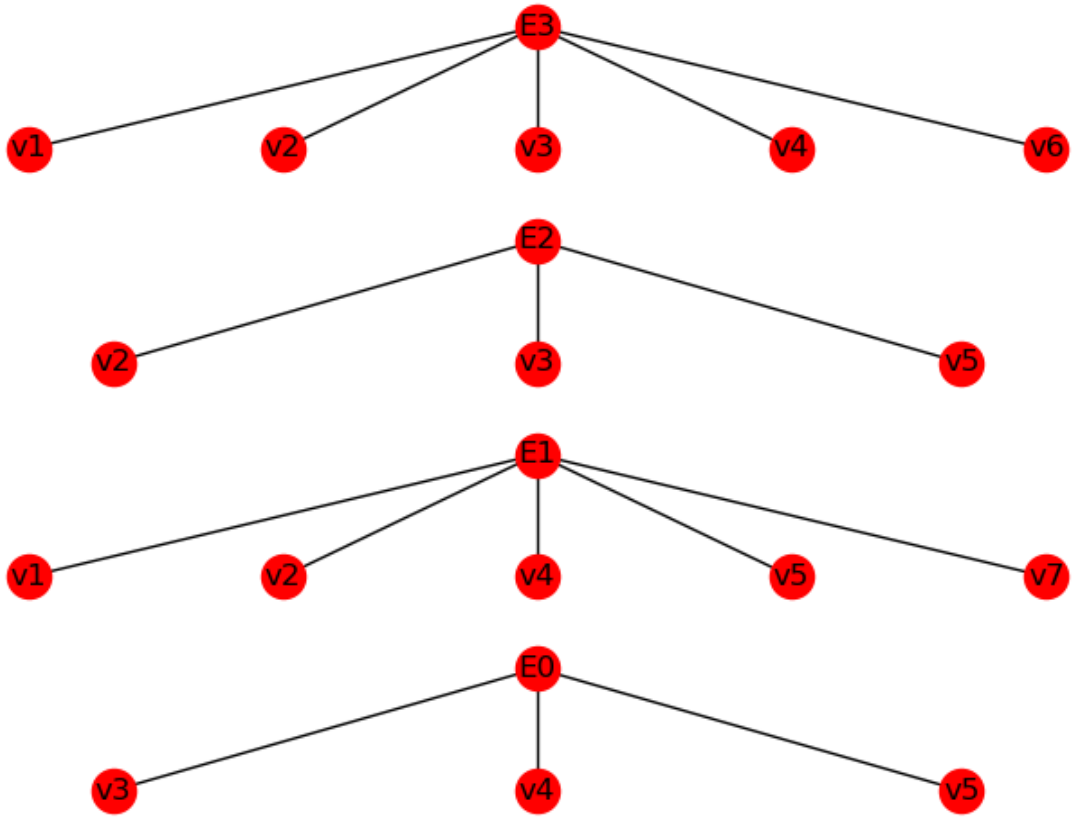

FIGURE 6 – Exemple d'affichage d'un graph dual

Divers

about Random

Comme demandé, nous avons brièvement regardé comment fonctionnait le pseudo aléatoire produit par la librairie random et plus particulièrement comment fonctionnait la méthode randint. À défaut d'avoir saisi l'intégralité de leur fonctionnement, nous avons pu comprendre que le tout reposait sur des masques et des shifts, ainsi que différentes manipulations, telle que la transformation d'un octet en un entier 64bits.

De plus, il est à noter que dans nos générations (d'arbres, par exemple) aléatoires, tous les arbres n'ont pas une probabilité identique d'être créé. En effet, d'après notre implémentation de la création des arbres : au plus un noeud aura été créé tôt, au plus il est probable qu'il ait un grand nombre de fils.

Bibliographie

[1] Joel. Hierarchy position, Aug 2018. https ://stackoverflow.com/revisions/29597209/9.

[2] Wikipedia. Tree traversal, Nov 2018. https ://en.wikipedia.org/wiki/Tree_traversal.

[3] Wikipedia. Chordal graph, Nov 2018. https ://en.wikipedia.org/wiki/Chordal_graph.

[4] Rose D. J. ; Tarjan R. E. ; Lueker G. S. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, pages 266–283, 1976.

[5] Wikipedia. Bron-kerbosch algorithm, Nov 2018. https ://en.wikipedia.org/wiki/Bron-Kerbosch_algorithmh.