

# INFO-F203 - Algorithmique 2: Projet 1

## Rapport

Pascal Tribel  
Noémie Muller

11 novembre 2018

### **Sommaire**

## Première partie

# Sous-arbre de poids maximum

## 1 Introduction

## 2 Choix d'implémentation

### 2.1 La classe "Tree"

La classe Tree a été implémentée sous forme d'un arbre récursif, où les enfants d'un noeud sont eux-mêmes un arbre, stockés dans une liste. Si cette liste est vide, on est alors confronté à une feuille. L'intérêt de cette implémentation est d'une part que le code est tout à fait compréhensible et lisible, et que la plupart des opérations se font avec une complexité raisonnable.

#### 2.1.1 La méthode somme

Cette méthode donne la valeur de la somme de tous les noeuds d'un arbre (c'est-à-dire la somme des sommes de ses sous-arbres). Cette opération de visite qu'une fois chaque noeud, et donc s'exécute avec une complexité de  $O(n)$  pour  $n$  le nombre de noeuds de l'arbre.

### 2.2 La fonction maxSubTree

Cette fonction détermine le sous-arbre maximum de l'arbre passé en paramètre. Sa complexité s'exprime en  $O(n^2)$  pour  $n$  le nombre de noeuds de l'arbre (en effet, chaque noeud n'est visité qu'une fois, pour calculer la somme du sous-arbre dont il est racine).

## 3 Conclusion

L'implémentation de cet exercice a été fait de manière à être la plus compréhensible possible, c'est pourquoi nous n'avons pas utilisé de modules externes. L'objectif était d'entièrement contrôler nos structures de données, en permettant ainsi de nous assurer de sa clarté et de son efficacité.

## Deuxième partie

# Les hypergraphes et hypertrees

## 4 Introduction

Pour cet exercice, comme pour le précédent, nous avons décidé d'implémenter nous-même les classes, les méthodes et les fonctions, au lieu de les emprunter à des modules. En effet, au-delà de l'intelligibilité du code écrit, cela nous a permis d'entièrement contrôler l'exécution de notre programme. Évidemment, ce choix comporte ses inconvénients : nous ne pouvons être certains d'avoir écrit les algorithmes les plus efficaces, et nous sommes limités quant aux représentations graphiques qui nous sont permises. Néanmoins, il nous a semblé que notre compréhension primait sur ces-dits inconvénients.

## 5 Choix d'implémentation

### 5.1 Les classes

#### 5.1.1 La classe `HyperNode`

Cette classe hérite de la classe `Node`, car un hyper-noeud est un noeud qui appartient à une hyper-arête. La complexité de toutes ses méthodes s'exprime en  $O(1)$ .

#### 5.1.2 La classe `HyperArete`

A l'initialisation d'un objet de cette classe, on va marquer dans chaque hyper-noeud de l'hyper-arête qu'il appartient à celle-ci. Aussi, la création d'un tel objet s'exécute avec une complexité exprimée en  $O(n)$ , pour  $n$  le nombre de noeuds qu'il contient. Il en va de même pour la fonction d'affichage. Toutes les autres méthodes s'exécutent en  $O(1)$ .

Le lecteur peut s'étonner de ne voir aucun setter pour cette classe. En effet, nous n'avons pas prévu que la classe serait utilisée en dehors du contexte que nous prévoyons, et de ce fait, un setter n'est pas requis.

#### 5.1.3 La classe `HyperGraphe`

Dernière classe de la famille des "hyper", elle est définie par les hyper-noeuds et les hyper-arêtes qu'elle contient. Ses méthodes d'expriment toutes en  $O(1)$ , sauf sont affichage, qui dépend de l'affichage de chacune de ses hyper-arêtes. Cette méthode d'affichage s'exprime donc avec une complexité de  $O(n * m)$ , pour  $n$  le nombre d'hyper-arêtes et  $m$  le nombre d'hyper-noeuds de l'hyper-graphe.

#### 5.1.4 La classe `Node`

#### 5.1.5 La classe `Graphe`

### 5.2 Les fonctions

#### 5.2.1 `incidenceGraph`

#### 5.2.2 `primalGraph`

#### 5.2.3 `dualGraph`

#### 5.2.4 `isCordal`

#### 5.2.5 `getMaximalClique`

#### 5.2.6 `getCycles`

#### 5.2.7 `isAlphaAcyclic`

#### 5.2.8 `isHyperTree`

## 6 Conclusion