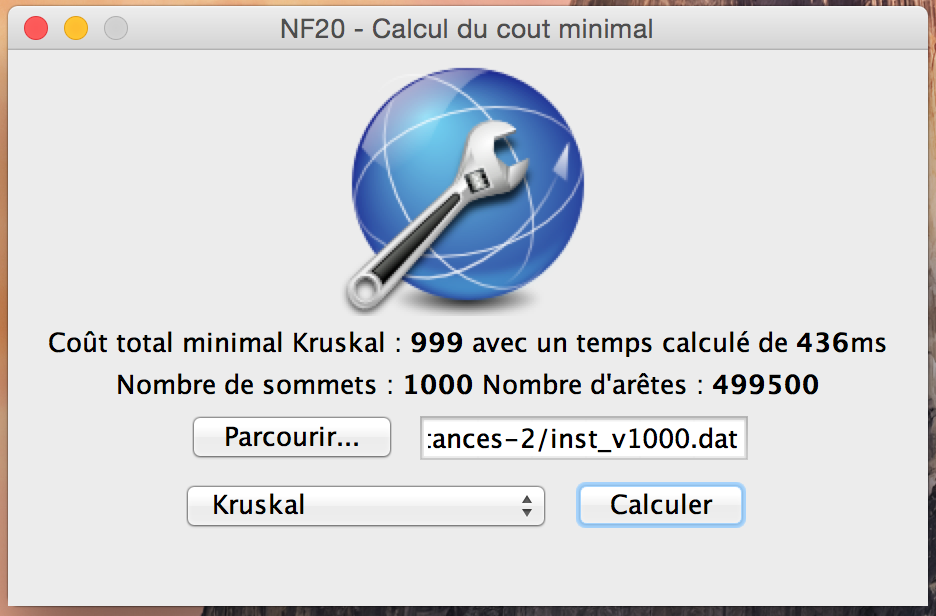


**Ali QUENAHAT**

**Olivier AFFIA**

Rapport NF20

Analyse de complexité des algorithmes d’arbres couvrants de poids minimal



**Semestre :Automne 2014**

**UV : NF20**

Sommaire

I. Analyses de la complexité théorique 2

A. Algorithme de Kruskal 2

B. Algorithme de Prim 3

C. Algorithme d’Elimination par cycles 5

II. Analyses expérimentales de la complexité 5

A. Algorithme de kruskal : 5

# I. Analyses de la complexité théorique

## Algorithme de Kruskal

L’algorithme de Kruskal permet de trouver un arbre couvrant de poids minimal, c’est-à-dire un graphe partiel connexe de poids minimum car les sommets sont les mêmes dont les coûts sont nuls ou positifs. Il permet cela à partir d’un graphe connexe et non orienté. L'algorithme range dans un premier temps par ordre de coûts croissant les arêtes, puis retire les arêtes toujours selon le même ordre et enfin les ajoutes à l'arbre couvrant minimal tant qu’il ne produit pas un cycle dans cet arbre. La meilleure complexité théorique de l’algorithme qu’il est possible d’obtenir est : .

Voici l’implémentation de l’algorithme :

TreeSet<Arete> aretes = **new** TreeSet<Arete>();

**for** (**int** i = **0**; i < nbAretes; i++) {

aretes.add(**new** Arete(graphe[i][**0**], graphe[i][**1**], graphe[i][**2**]));

}

KruskalAretes arbreFinal = **new** KruskalAretes();

**for** (Arete arete : aretes) {

arbreFinal.rajouterArete(arete);

}

Les méthodes appelées sont :

**class** **KruskalAretes** {

Vector<HashSet<Integer>> sommetGroupes = **new** Vector<HashSet<Integer>>();

TreeSet<Arete> kruskalAretes = **new** TreeSet<Arete>();

**public** TreeSet<Arete> **getAretes**() {

**return** kruskalAretes;

}

HashSet<Integer> **getSommetGroupe**(**int** sommetDepart) {

**for** (HashSet<Integer> sommetGroupe : sommetGroupes) {

**if** (sommetGroupe.contains(sommetDepart)) {

**return** sommetGroupe;

}

}

**return** **null**;

}

**public** **void** **rajouterArete**(Arete arete) {

**int** sommetDepart = arete.SommetDepart();

**int** sommetArrive = arete.SommetArrive();

HashSet<Integer> sommetGroupeDepart = getSommetGroupe(sommetDepart);

HashSet<Integer> sommetGroupeArrive = getSommetGroupe(sommetArrive);

**if** (sommetGroupeDepart == **null**) {

**if** (sommetGroupeArrive == **null**) {

HashSet<Integer> NouveauGroupeSommet = **new** HashSet<Integer>();

NouveauGroupeSommet.add(sommetDepart);

NouveauGroupeSommet.add(sommetArrive);

sommetGroupes.add(NouveauGroupeSommet);

kruskalAretes.add(arete);

} **else** {

sommetGroupeArrive.add(sommetDepart);

kruskalAretes.add(arete);

}

} **else** {

**if** (sommetGroupeArrive == **null**) {

sommetGroupeDepart.add(sommetArrive);

kruskalAretes.add(arete);

} **else** **if** (sommetGroupeDepart != sommetGroupeArrive) {

sommetGroupeDepart.addAll(sommetGroupeArrive);

sommetGroupes.remove(sommetGroupeArrive);

kruskalAretes.add(arete);

}

}

}

}

Nous avons jugé nécessaire de mettre notre code dans ce rapport, au moins pour ce premier algorithme pour que vous puissiez comprendre notre raisonnement pour effectuer le calcul. Donc après le calcul de la complexité de notre algorithme de Kruskal, nous obtenons :



En développant et en simplifiant nous obtenons une complexité quadratique : O(n²)

Il faut savoir que les opérations add, remove, contains exécutées en un temps log(n) lors de l’utilisation d’un TreeSet, et sont exécutées en un temps constant pour un HashSet. De plus notre résultat s’explique par le fait que l’on appelle la méthode rajouterArete() dans une boucle dans la principale méthode, or cette méthode fait elle même appel à la méthode getSommetGroupe() qui contient une boucle qui est de complexité O(nbAretes). Au final, nous avons là une boucle imbriquée dans une autre et en négligeant log(n) et n devant n², ce qui nous explique la complexité quadratique théorique trouvée de notre algorithme. Aussi, pour chaque cas, nous nous sommes placé dans le pire cas possible, et c’est ce que nous allons faire pour les algorithmes qui suivent.

## Algorithme de Prim

L’algorithme de Prim permet de trouver un arbre couvrant de poids minimal tout comme l’algorithme de Kruskal, cependant il ne fonctionne pas pareil. Il choisit un sommet et crée un arbre à partir de ce sommet en ajout les arêtes de poids le plus faible.

Nous obtenons après calcul :



Toujours comme pour Kruskal, nous avons une complexité quadratique après un développement et une simplification : O(n²)

Cela s’explique par le fait qu’on a une boucle imbriquée dans un autre là encore, sachant que la boucle imbriquée a une complexité de O(n), cela explique qu’avec la boucle qui l’imbrique nous obtenons une complexité quadratique. Encore une fois, nous avons négligé n devant n².

## Algorithme d’élimination par cycles

La méthode d’élimination par cycles permet elle aussi de trouver un arbre couvrant de coût minimal. Il consiste, lorsque l’on forme un cycle en rajoutant un sommet, à « casser » ce cycle en y enlevant l’arête de poids le plus fort.

Nous trouvons comme complexité théorique en utilisant toujours la même méthode de calcul : O(n^4)

Nous ne mettrons pas l’implémentation de l’algorithme dans ce rapport à cause de sa longueur.

# II. Analyses expérimentales de la complexité

Maintenant que nous avons analysé les complexités théoriques, nous allons analyser les complexités expérimentales afin de voir l’écart entre ces deux.

Nous avons utilisé un MacBook PRO pour notre expérience avec un processeur Intel Core i5 bicœur à 2,6 GHz (Turbo Boost jusqu’à 3,1 GHz). Le processeur a une puissance de 3,1Ghz. Pour pouvoir comparer les temps nous devions trouver une valeur qui même si approximative définit le nombre moyen d’instructions que notre processeur fait par seconde. Nous avons donc trouvé sur internet qu’il s’agit pour notre cas de 42 200 millions d’instructions par secondes. Le langage de programmation utilisé est JAVA 6.0. Nous avons choisis un tableau comme structure de données pour l’algorithme de Prim, un TreeSet pour Kruskal.

Sachant que les complexités sont de O(n^2) pour les deux premiers algorithmes et de O(n^4) pour celui de l’élimination par cycles, nous avons modélisé l’écart entre les mesures théoriques et expérimentales à l’aide de graphiques grâce à ce qu’on a trouvé dans la partie I. Nous allons donc vous présenter nos mesures pour chaque algorithmes.

## Algorithme de Kruskal

|  |  |  |
| --- | --- | --- |
| Nombre de sommets | Complexité expérimentale (en ms) | Complexité théorique (en ms) |
| 100 | 5 | 0,23 |
| 200 | 11 | 0,94 |
| 300 | 23 | 2,1 |
| 400 | 47 | 3,7 |
| 500 | 69 | 5,9 |
| 600 | 99 | 8,5 |
| 700 | 164 | 11,6 |
| 800 | 202 | 15,1 |
| 900 | 256 | 19,1 |
| 1000 | 481 | 23,6 |

## Algorithme de Prim :

|  |  |  |
| --- | --- | --- |
| Nombre de sommets | Complexité expérimentale (en ms) | Complexité théorique (en ms) |
| 100 | 6 | 0,23 |
| 200 | 19 | 0,94 |
| 300 | 67 | 2,1 |
| 400 | 158 | 3,7 |
| 500 | 346 | 5,9 |
| 600 | 608 | 8,5 |
| 700 | 963 | 11,6 |
| 800 | 1463 | 15,1 |
| 900 | 2116 | 19,1 |
| 1000 | 2830 | 23,6 |

## Algorithme d’élimination par cycles

|  |  |  |
| --- | --- | --- |
| Nombre de sommets | Complexité expérimentale (en ms) | Complexité théorique (en ms) |
| 100 | 53 | 2,36 |
| 200 | 638 | 37,91 |
| 300 | 2761 | 191,94 |
| 400 | 8162 | 606,63 |
| 500 | 18946 | 1481 |
| 600 | 39319 | 3071 |

Nous avons donc réalisé un tableau récapitulant nos tests expérimentaux, pour chaque algorithme, ainsi que le temps que l'on devrait trouver théoriquement.

Il faut savoir que pendant cette expérience, nous avons eu l’idée de faire une moyenne des valeurs afin d’avoir une valeur un peu plus précise même si nous pensons que cela est négligeable devant l’écart.

L’écart obtenu est dû à plusieurs facteurs. En effet selon le nombre de logiciels qui tournent, le système d’exploitation, le processeur et le langage impliquent que les temps soient soumis à un coefficient qui explique donc l’écart. Mais aussi un autre facteur important est l’approximation faite pour les calculs que nous avons détaillé dans la première partie. Nous pensons aussi qu’un autre facteur intervient, en effet, le nombre d’instructions par secondes que nous avons trouvé sur internet est une mesure approximative car cette information n’est pas donnée par l’entreprise fabriquant le processeur.

Toutefois on peut dire que les allures des courbes ont au final la même tendance malgré toutes les approximations qui ont été faites mais cela est encore plus visible lors d’une adaptation de l’échelle pour la complexité théorique.