Florent Lucet Andrea Duhamel
Marie-Florence Reveneau

Projet de NF20 :
Prim, Kruskal et calcul de diamètre

Dans le cadre du cours NF20, nous avons été amenés à effectuer l'analyse et la conception des algorithmes de Prim et Kruskal, ainsi que du calcul du diamètre d'un graphe. Nous avions alors pour objectif personnel de profiter de cette occasion pour découvrir un nouveau langage : OCaml. Ce langage a été choisi car il s'agit d'un langage fonctionnel qui a été conçu spécialement pour le calcul mathématique, notamment pour le calcul de la complexité algorithmique. Au fur et à mesure du projet, des obstacles et des difficultés sont apparus, entre autre liés à l'apprentissage d'un nouveau langage. Nous avons ainsi été amenés à réaliser nos algorithmes en langage Java (que nous avons étudié à l'UTT), en parallèle avec OCaml.

I) Analyse de complexité des algorithmes de Prim et Kruskal

1) Algorithme de Prim

a) Avec le langage OCaml

Cet algorithme de Prim est écrit en langage OCaml. Il s'agit d'un langage fonctionnel utilisant, comme son nom l'indique, une succession de fonctions (souvent récursives) afin d'obtenir le résultat voulu.

Dans les algorithmes de Prim et du calcul du diamètre, on stocke un graphe grâce à une liste de tuples, tels qu'un tuple, qui représente en fait une arête, est caractérisé par :
tuple = (sommet1, sommet2, poids).

Les fonctions utilisées de complexité non-négligeable sont les suivantes :

En nommant x le nombre d'éléments d'une liste :

La fonction count a pour complexité O(x) ;
La fonction min a pour complexité O(x) ;
La fonction estDans a pour complexité O(x) ;
La fonction addSommet a pour complexité O(x) ;
La fonction possibilite a pour complexité O(x) ;
La fonction listePossible a pour complexité O(x^2) ;
La fonction addArbre a pour complexité O(x+x.x.(x^2)).

En notant maintenant n le nombre de sommets et m le nombre d'arêtes du graphe, nous avons :
Complexité de Prim = n . complexité de addArbre = n . O(n+n.m.m^2) = O(n^2+n.m^3)

Complexité de cet algorithme de Prim : O(n^2+n.m^3).

La complexité théorique optimale de l'algorithme de Prim qui utilise un tas de Fibonacci, est de O(m+nlog(n)), ce qui est bien inférieur à la complexité de notre algorithme.

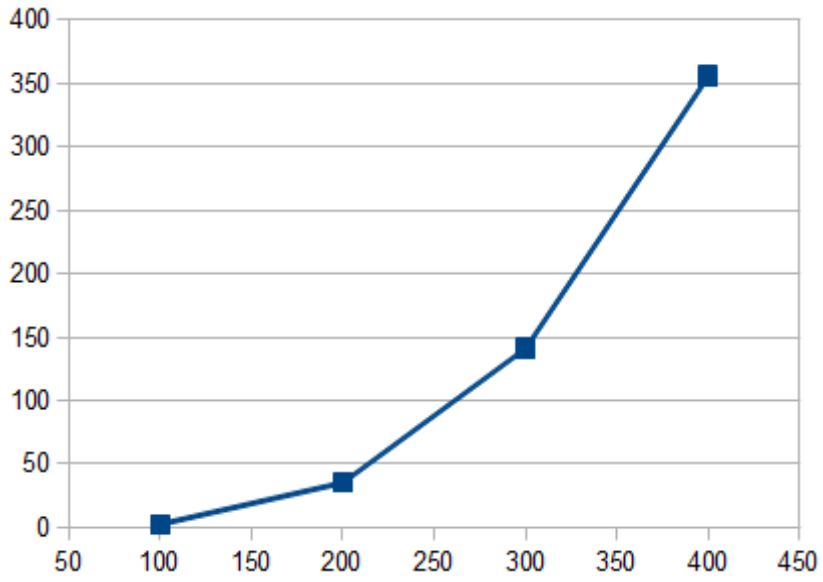
b) Avec le langage Java

L'algorithme de Prim a aussi été réalisé en langage Java (version 7), grâce à une recherche en largeur adaptée et couplée à une matrice d'adjacence.

Une complexité théorique ayant déjà été calculée dans la version écrite en OCaml, celle-ci n'a pas été recalculée dans la version en Java, bien que différente.

Dans un contexte expérimental, des tests ont été réalisés pour pouvoir dessiner la courbe de la complexité expérimentale. Celle-ci est représentée par le temps de calcul en fonction de la taille des instances. En voici les résultats :
* Instance de 100 : 2,527 secondes.
* Instance de 200 : 35,319 secondes.
* Instance de 300 : 141,104 secondes.
* Instance de 400 : 355,715 secondes.

En voici la courbe :
Abscisse x : taille de l'instance.
Ordonnée y : temps (en secondes).

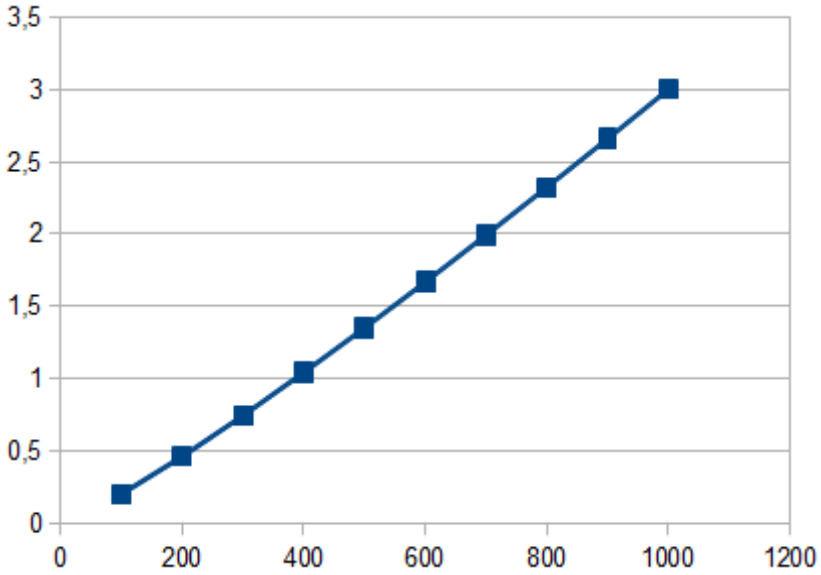


2) Algorithme de Kruskal

L'algorithme de Kruskal a été réalisé en langage Java (version 7), grâce à la structure de données Union-Find couplée avec deux optimisations de complexité : pondération du graphe et compression des chemins.

La complexité de l'algorithme de Kruskal est donnée par la complexité de sa méthode de tri, c'est-à-dire O(m log m), où m représente le nombre d'arêtes.

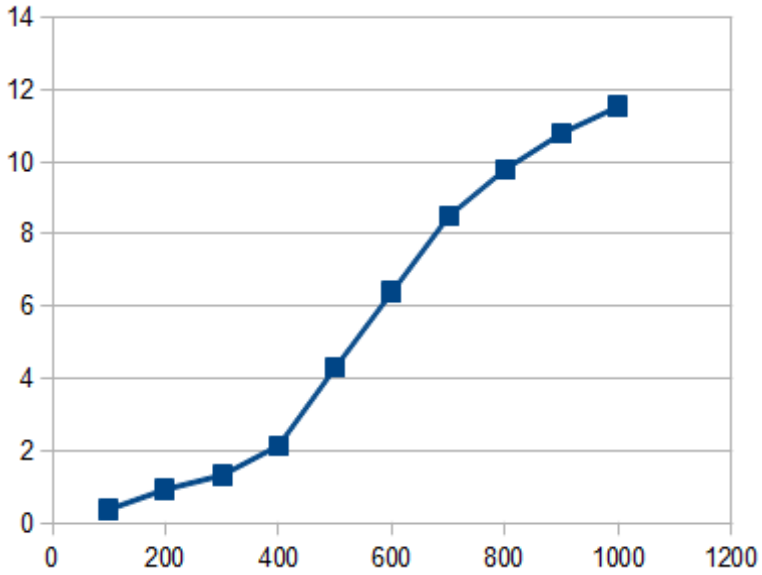
Voici la courbe de la complexité théorique :
Abscisse x : taille de l'instance.
Ordonnée y : temps (en secondes).



Dans un contexte expérimental, des tests ont été réalisés pour pouvoir dessiner la courbe de la complexité expérimentale. Celle-ci est représentée par le temps de calcul en fonction de la taille des instances. En voici les résultats :

* Instance de 100 : 0,371 seconde.
* Instance de 200 : 0,924 seconde.
* Instance de 300 : 1,316 seconde.
* Instance de 400 : 2,139 secondes.
* Instance de 500 : 4,296 secondes.
* Instance de 600 : 6,393 secondes.
* Instance de 700 : 8,489 secondes.
* Instance de 800 : 9,787 secondes.
* Instance de 900 : 10,791 secondes.
* Instance de 1000 : 11,529 secondes.

En voici la courbe :
Abscisse x : taille de l'instance.
Ordonnée y : temps (en secondes).



En comparant les deux courbes, nous nous apercevons que la courbe théorique atteint 3 secondes de temps d'exécution à l'instance de 1000. Ce même temps est déjà atteint à l'instance de 400 sur la courbe expérimentale. Ce décalage peut s'expliquer par l'environnement d'exécution, l'ordinateur sur lequel nous avons testé l'algorithme étant un ordinateur peu puissant. Cependant, même si nous pouvons observer un écart, les courbes restent relativement proches.

II) Algorithme de calcul du diamètre d'un graphe

Pseudo-code :

Note : *premier(tuple)* renvoie la première valeur du tuple, qui correspond à un sommet d'une arête ;
deuxieme(tuple) renvoie la deuxième valeur du tuple, qui correspond à l'autre sommet de l' arête ;
troisieme(tuple) renvoie la troisième valeur du tuple, qui est le poids de l'arête.

On rappelle aussi qu'un graphe (et donc un arbre) est une succession de tuples.

Soit la fonction récursive *diametre*. Cette fonction applique pour chaque point de l'arbre la fonction *distMax* suivante :

Soit la fonction récursive *distMax (arbre, racine)* = // on part d'un sommet racine
Si l'arbre est vide, alors renvoyer (0,0,0)
Si arbre= tuple+resteTuples (c'est-à-dire si l'arbre n'est pas vide) alors
Si cette racine est un des sommets de l'arête correspondant à tuple alors
Si cette arête a un poids plus élevé que ceux des autres tuples
(récursivité) alors
ajouter (distMax (reste, autre sommet du tuple)) (troisieme tuple)
Sinon ajouter (distMax (reste, autre sommet du tuple)) (troisieme (distMax (reste racine)))
else distMax (reste,racine);; // on ajoute pour chaque sommet, le plus court chemin de ce sommet à chacun des autres.

Complexité :

L'algorithme du calcul du diamètre, tout comme l'algorithme de Prim, a été écrit en OCaml. La fonction principale *diamètre,* qui utilise la fonction *prim* de l'algorithme de Prim de complexité O(n^2+n.m^3) comme nous l'avons vu précédemment, et à laquelle s'ajoute la fonction *distMax* de complexité O(n), a pour complexité :

O(n).O(n^2+n.m^3) = O(n^4+(nm)^3)

Complexité de cet algorithme de calcul du diamètre : O(n^4+(nm)^3).

Conclusion

Ce projet nous a permis de mettre en pratique les connaissances acquises à l'UTT, aussi bien dans l'UV NF20 que dans l'UV LO02 (programmation en Java). Il nous a également permis de nous rapprocher d'une situation proche de celle qui nous attend dans le monde de l'entreprise en tant qu'ingénieur : travail en équipe, délais, recherches supplémentaires, etc. À présent, grâce à l'UV NF20, il nous sera plus facile d'optimiser nos algorithmes et de chercher les structures de données les plus adaptées à un cas donné. De plus, ce projet nous aura à la fois permis de découvrir un nouveau paradigme de programmation et un nouveau langage, avec OCaml, langage fonctionnel.