**[NF20] Rapport de projet**

Algorithmes de plus court chemin et analyse de complexité

HALOUI Amine / JALOUZET Jérémie / XU Jiahuan

Présentation des implémentations et de l'analyse de complexité des algorithmes de Dijkstra et de Bellman-Ford

# Sommaire

[Introduction 2](#_Toc313720449)

[I) Analyses théoriques de complexité 3](#_Toc313720450)

[1) Algorithme de Dijkstra 3](#_Toc313720451)

[2) Algorithme de Bellman-Ford 4](#_Toc313720452)

[II) Analyses expérimentales de complexité 6](#_Toc313720453)

[Conclusion 8](#_Toc313720454)

# Introduction

Blablabla...

# Analyses théoriques de complexité

## Algorithme de Dijkstra

L’algorithme de **Dijkstra** permet de trouver le plus court chemin dans un graphe donné. Ainsi, la complexité théorique de l’algorithme est la suivante : (). Cette dernière est appelée complexité logarithmique et correspond à l’une des meilleures complexités que l’on peut avoir.

Voici une implémentation en *Java* de l’algorithme de Dijkstra :

|  |
| --- |
| 01 **public int**[] doAlgorithm\_Dijkstra(**int**initialNode) { 02         **int**[] minimalCosts = **new int**[numberOfNodes]; 03         **boolean**[] visitedNodes = **new boolean**[numberOfNodes]; 04         **for**(**int**i = 0; i < numberOfNodes; i++) { 05             minimalCosts[i] = INFINITE\_COST; 06             visitedNodes[i] = **false**; 07         } 08         minimalCosts[initialNode] = 0; 09         **int**min = INFINITE\_COST; 10         **int**x = 0; 11         **int**y = 0; 12  13         **for**(**int**k = 0; k < numberOfNodes; k++) { 14             min = INFINITE\_COST; 15             **for**(y = 0; y < numberOfNodes; y++) { 16                 **if**((!visitedNodes[y]) && (minimalCosts[y] < min)) { 17                     x = y; 18                     min = minimalCosts[y]; 19                 } 20             } 21             visitedNodes[x] = **true**; 22             **int**[] successors = findSuccessors(x); 23             **for**(**int**i = 0; i < successors.length; i++) { 24                 y = successors[i]; 25                 **if**(min + listOfArcs[x][y] < minimalCosts[y]) { 26                     minimalCosts[y] = min + listOfArcs[x][y]; 27                 } 28             } 29         } 30         **return**minimalCosts; 31     } 32  33 **private int**[] findSuccessors(**int**node) { 34   ArrayList<Integer> arrayList = **new**ArrayList<Integer>(); 35         **for**(**int**i = 0; i < listOfArcs.length; i++) { 36             **if**(listOfArcs[node][i] != INFINITE\_COST) { 37                 arrayList.add(i); 38             } 39         } 40         **return**copyListIntoArray(arrayList); 41     } |

Ainsi, à partir de la ligne 1 à la ligne 11, nous avons la complexité suivante : (1), sauf entre la ligne 4 et 7 où nous avons la complexité suivante : (). A partir de la ligne 13, nous entrons dans le cœur de l’algorithme :

En simplifiant, nous trouvons la complexité suivante : (), ce qui correspond à une complexité cubique. Cela s’explique par l’appel à la méthode *findSuccessors*. Tout d'abord, celle-ci se situe dans une boucle. De plus, celle-ci fait également appel à une boucle dans laquelle on appelle la méthode *add*, via un objet de type *ArrayList* (qui possède une complexité[[1]](#footnote-1) en ()). Enfin, le coût de l’appel à la méthode *copyListIntoArray* est négligeable.

## Algorithme de Bellman-Ford

Voici une implémentation en *Java* de l'algorithme de **Bellman-Ford** :

|  |
| --- |
| 01 **public int**[] doAlgorithm\_Bellman(**int**initialNode) { 02         **int**[] minimalCosts = **new int**[numberOfNodes]; 03         **for**(**int**i = 0; i < numberOfNodes; i++) { 04             minimalCosts[i] = INFINITE\_COST; 05         } 06         minimalCosts[initialNode] = 0; 07         **boolean**found = **false**; 08         **int**k = 0; 09         **do**{ 10             k++; 11             found = **false**; 12             **for**(**int**y = 0; y < minimalCosts.length; y++) { 13                 **int**[] predecessor = findPredecessors(y); 14                 **for**(**int**i = 0; i < predecessor.length; i++) { 15                     **int**x = predecessor[i]; 16                     **if**(minimalCosts[x] + listOfArcs[x][y] < minimalCosts[y]) { 17                         minimalCosts[y] = minimalCosts[x] + listOfArcs[x][y]; 18                         found = **true**; 19                     } 20                 } 21             } 22         } **while**(found); 23         **return**minimalCosts; 24     } |

A partir de la ligne 2 et jusque la ligne 8, la complexité est en (), sauf pour les lignes 3 à 5 où la complexité est en(). A partir de la ligne 9, nous entrons dans le cœur de l’algorithme :

En simplifiant, nous trouvons la complexité suivante : (), ce qui correspond à une complexité cubique. Les raisons de cette complexité sont les mêmes que pour l’algorithme précédent. En effet, la méthode *findPredecessors (*qui est presque la même méthode que *findSuccessors)* est coûteuse, car elle combine une boucle et un appel à une méthode dont la complexité est en (). Cela nous donne une complexité en (). Or, cette méthode est aussi dans une boucle, ce qui fait apparaître une complexité en ().

Nous souhaitons présenter une autre implémentation de l’algorithme de Bellman-Ford, qui affiche une complexité quadratique :

|  |
| --- |
| 01 **public void**algorithmeBellman() 02 {  03  04   **for**(**int**i = 0; i < 2; i++) 05   { 06     **for**(Sommet sommet : grapheEnConstruction.getListeSommets()) 07     { 08       **for**(Arc arc : grapheConstruit.getListeArcsParSommet(sommet)) 09       { 10         **if**(arc.getSommetDestination().getPoids() >  11           (arc.getSommetOrigine().getPoids() + arc.getPoids())) 12         { 13           arc.getSommetDestination().setPoids( 14             arc.getSommetOrigine().getPoids() + arc.getPoids()); 15           arc.getSommetDestination().setPredecesseur(arc.getSommetOrigine()); 16         } 17       } 18     } 19   } 20 } 21  22 **public**ArrayList<Arc> getListeArcsParSommet (Sommet sommet) 23 { 24   LinkedList<Arc> listeArcsParSommetsTemporaire = **new**LinkedList<Arc>(); 25   **for**(Arc arc : sommet.getListeArcConnecte()) 26   { 27     listeArcsParSommetsTemporaire.add(arc); 28   } 29      30   ArrayList<Arc> listeArcsParSommets =  **new**ArrayList<Arc>(listeArcsParSommetsTemporaire); 31      32   **return**listeArcsParSommets; 33 } |

La complexité algorithmique est la suivante :

Après simplification, nous obtenons une complexité en () qui est permise via deux boucles imbriquées. Le seul facteur pouvant influer sur la complexité est situé dans la deuxième boucle, par la méthode getListArcsParSomme. Or, cette méthode fait appel à la structure de données *LinkedList* dont la complexité[[2]](#footnote-2) est en () pour l’ajout d’un élément. De plus, cette structure est appelée pour chacun des arcs, ce qui permet d’obtenir une complexité en (). Enfin, cette structure est appelée pour chacun des sommets, ce qui nous permet d’afficher une complexité en ().

Il est également important de noter que nous étudions la complexité asymptotique. Dès lors, nous présentons systématiquement, via les formules mathématiques, des sommes sur l’ensemble des sommets. Or, le code Java effectue certaines fois des boucles sur les arcs. Cette simplification est importante car cela permet d’indiquer que nous sommes dans le pire des cas, celui où chaque sommet est relié à tous les autres sommets.

# Analyses expérimentales de complexité

Après avoir implémenté les algorithmes de Dijkstra et Bellman-Ford en *Java*, et analysé théoriquement la complexité de ceux-ci, nous allons procéder à une analyse expérimentale.

La machine ayant servi pour nos tests est un ordinateur portable comportant un processeur *Intel® Core™ i7-2670QM*. Il s'agit d'un processeur disposant de **4** coeurs, ayant chacun une puissance de **2,20 GHz** *(soit 2,20 x 109 Hz)*. Cependant, d'après nos tests, il semble que notre application n'utilise qu'un seul cœur.

Au final, la machine de test avait donc une fréquence de **2,20 x 109** **Hz**.

Comme nous avons au préalable trouvé que nos implémentations des algorithmes de Dijstra et Bellman-Ford ont toutes les deux une complexité algorithmique de (), nous pouvons calculer combien de temps *devrait* prendre le calcul du plus court chemin selon un certain nombre de sommets dans le graphe. Nous avons donc réalisé un tableau récapitulant nos tests expérimentaux, pour chaque algorithme, ainsi que le temps que l'on devrait trouver théoriquement.

Les calculs étaient trop rapides pour pouvoir mesurer avec précision le temps nécessaire pour effectuer un algorithme. C'est pourquoi, pour chaque expérience (avec un certain nombre de sommets dans le graphe), nous avons fait une boucle pour réaliser chaque algorithme **1000** **fois**. Suite à cela, nous avons donc divisé le temps obtenu par 1000, et cela nous donne ainsi le temps moyen pour réaliser **1 fois** l'algorithme.

Il est à noter qu'au cours de nos expérimentations, les temps obtenus variaient beaucoup. Les mesures présentées ici ne sont pas donc forcément très précises, car d'autres processus interfèrent avec celui de notre application : système d'exploitation, applications en tâches de fond...Tous ces processus prennent aussi de la ressource processeur. Pour obtenir les meilleurs résultats possibles, nous avons donc, pour chaque expérience, lancé la boucle de l'algorithme (qui effectue déjà 1000 fois l'algorithme) plusieurs fois, et fait une moyenne des temps obtenus.

Enfin, nous avons aussi fait nos expériences sur plusieurs types de graphes : **graphe non-orienté**, et **graphe orienté**.

Pour rappel, les structures de données que nous avons utilisé sont :

* des matrices (tableaux à 2 dimensions) contenant des entiers
* des *ArrayList* *(objets Java qui permettent de faire des tableaux d'objets dynamiques)* d'entiers

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre de sommets dans le graphe | Algorithme de Dijkstra | Algorithme de Bellman-Ford | Temps théorique |
| 100 | 0,22 ms | 0,81 ms | 0,45 ms |
| 200 | 0,90 ms | 3,49 ms | 3,64 ms |
| 300 | 2,15 ms | 7,95 ms | 12,27 ms |
| 400 | 3,74 ms | 18,41 ms | 29,09 ms |
| 500 | 5,71 ms | 28,86 ms | 56,82 ms |
| 600 | 8,38 ms | 45,20 ms | 98,18 ms |
| 700 | 11,29 ms | 62,40 ms | 155,91 ms |
| 800 | 14,98 ms | 90,50 ms | 232,73 ms |
| 900 | 19,65 ms | 93,60 ms | 331,36 ms |
| 100 | 23,56 ms | 124,80 ms | 454,54 ms |

On peut remarquer qu'il y a de grands écarts entre les complexités théoriques et expérimentales des algorithmes. De plus, les résultats des expérimentations sont meilleurs que ceux attendus théoriquement, ce qui devrait être impossible. Cela peut être dû à plusieurs choses :

* Tout d'abord, les complexités théoriques se basent sur le pire des cas, c'est-à-dire si le graphe est dense, et que tous les sommets sont reliés entre eux. Ce n'est pas le cas dans les fichiers d'instance utilisés.
* Ensuite, pour calculer les résultats que l'on doit obtenir théoriquement, nous nous sommes servis de la fréquence du processeur, en GHz. Ce nombre ne correspond pas forcément au nombre d'instructions par seconde réellement effectué par la machine. En fait, il faut prendre le **MIPS** *(Million d'Instructions Par Seconde)* du processeur, pour avoir un résultat beaucoup plus proche de la réalité. Cependant, il s'agit de données difficiles à trouver, car rarement divulguées par le constructeur du processeur.

# Conclusion

Blablabla...

1. Programmer en java, 5ème édition, Claude Delannoy, page : 623 [↑](#footnote-ref-1)
2. Programmer en Java, 5ème édition, page : 617 [↑](#footnote-ref-2)