**Rapport NF20**

Algorithmique et complexité.

Auteurs

Présentation des implémentations des algorithmes de Dijkstra et de Bellman-Ford ainsi qu’une analyse de leurs complexités.

Sommaire

Analyse de la complexité : 3

Présentation : 3

Algorithme de Disjkstra : 3

Algorithme de Bellman-Ford : 4

# Analyse de la complexité :

## Présentation :

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.

## Algorithme de Disjkstra :

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 *findSuccessor*, effectivement, celle-ci se situe dans une première boucle, de plus, celle-ci fait également appelle 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 :

|  |
| --- |
| 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     } |

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, effectivement, la méthode *findPredecessor* qui est la même méthode que *findSuccessor,* est coûteuse car elle combine une boucle et un appel à une méthode dont la complexité est en (), ce qui correspond à 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 influé sur la complexité est situé dans la deuxième boucle par la méthode *getListArcsParSommets* 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 appelé 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.

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