[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

I) Analyses théoriques de complexité

1) 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 : $O(Log(N))$. 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 et jusque ligne 11, nous avons la complexité suivante : $6 * O(1)$, sauf entre la ligne 4 et 7 où nous avons la complexité suivante : $O(N)$. A partir de la ligne 13, nous entrons dans le cœur de l'algorithme :

$$\sum_{k=0}^{Sommets} (O(1) + \sum_{y=0}^{Sommets} \big(4 * O(N)\big) + 2 * O(1)$$
$$+ \sum_{g=0}^{Sommets} \big(O(1) + O(N)\big) + O(1) + \sum_{v=0}^{Sommets} (O(1)) + \sum_{u=0}^{Sommets} (6 * O(1)))$$

En simplifiant, nous trouvons la complexité suivante : $O(N^3)$, 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] en $O(N)$). Enfin, le coût de l'appel à la méthode *copyListIntoArray* est négligeable.

2) Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet également de déterminer le plus court chemin dans un graphe donné. Cependant, il est capable de prendre en compte les arcs de coûts négatifs. Cet algorithme possède une complexité théorique en $O(N * M)$, N et M représentant les sommets et les arcs du graphe.

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 $O(1)$, sauf pour les lignes 3 à 5 où la complexité est en $O(N)$. A partir de la ligne 9, nous entrons dans le cœur de l'algorithme :

$$\sum_{i=0}^{3}(2 * O(1) + \sum_{v=0}^{Sommets}\left(3 * O(1) + \sum_{g=0}^{Sommets}(O(1) + O(N)) + \sum_{u=0}^{Sommets}(O(1)) + \sum_{r=0}^{Sommets}(6 * O(1))\right))$$

[1] Programmer en java, 5ème édition, Claude Delannoy, page : 623.

En simplifiant, nous trouvons la complexité suivante : $O(N^3)$, 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 $O(N)$. Cela nous donne une complexité en $O(N^2)$. Or, cette méthode est aussi dans une boucle, ce qui fait apparaître une complexité en $O(N^3)$.

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 :

$$\sum_{i=0}^{2} * \left(\sum_{n=0}^{Sommets} (2 * O(N) + \sum_{g=0}^{Sommets} \left(11 * O(1)\right)\right))$$

Après simplification, nous obtenons une complexité en $O(N^2)$ qui est permise via deux boucles imbriquées. Le seul facteur pouvant influer sur la complexité est situé dans la deuxième boucle, par la méthode *getListeArcsParSommet*. Or, cette méthode fait appel à la structure de données *LinkedList* dont la complexité[2] est en $O(1)$ pour l'ajout d'un élément. De plus, cette structure est appelée pour chacun des arcs, ce qui permet d'obtenir une complexité en $O(N)$. Enfin, cette structure est appelée pour chacun des sommets, ce qui nous permet d'afficher une complexité en $O(N^2)$.

[2] Programmer en Java, 5ème édition, Claude Delannoy, page : 617.

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.

II) 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 10^9 Hz)*. Cependant, d'après nos tests, il semble que notre application n'utilise qu'un seul cœur. Aussi, la fréquence d'horloge (en Hz) ne permet pas de savoir le nombre d'instructions par seconde qu'effectue réellement le processeur. Pour cela, nous avons trouvé un benchmark indiquant que ce processeur fait en moyenne : **101 583 MIPS** *(millions d'instructions par secondes)*, soit **101,583 x 10^9 instructions par seconde**.

Comme nous avons au préalable trouvé que nos implémentations des algorithmes de Dijstra et Bellman-Ford ont toutes les deux une complexité algorithmique de $O(N^3)$, 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.

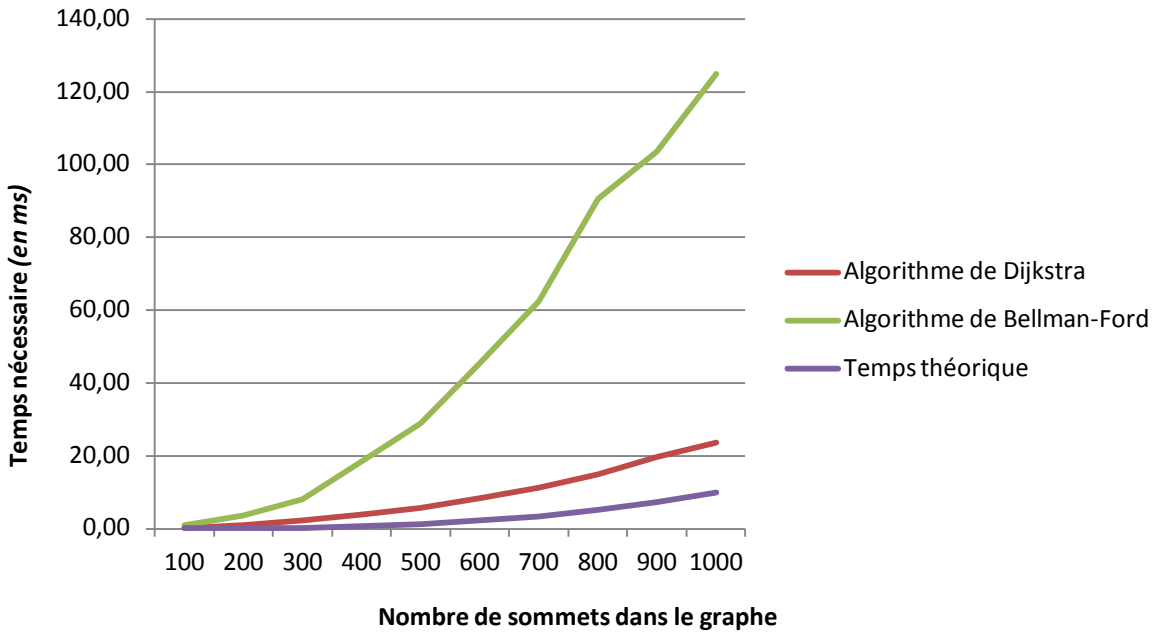
Il est à noter qu'au cours de nos expérimentations, les temps obtenus variaient beaucoup. Les mesures présentées ici ne sont pas donc forcément très précises, car d'autres processus interfèrent avec celui de notre application : système d'exploitation, applications en tâches de fond...Tous ces processus 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	0,0098 ms
200	0,90 ms	3,49 ms	0,079 ms
300	2,15 ms	7,95 ms	0,26 ms
400	3,74 ms	18,41 ms	0,63 ms
500	5,71 ms	28,86 ms	1,23 ms
600	8,38 ms	45,20 ms	2,13 ms
700	11,29 ms	62,40 ms	3,38 ms
800	14,98 ms	90,50 ms	5,04 ms
900	19,65 ms	103,60 ms	7,18 ms
100	23,56 ms	124,80 ms	9,84 ms



On peut tout d'abord remarquer que l'algorithme de Bellman-Ford est beaucoup plus lent que l'algorithme de Dijkstra. Il y a aussi des écarts entre les complexités théoriques et expérimentales des algorithmes :

- La première raison qui explique ces écarts est que la complexité théorique est la complexité idéale or nos implémentations ne sont pas forcément les meilleures malgré le temps assez important que nous leurs avons accordées et dépendent des structures de données choisies, des boucles placées ou encore du raisonnement de l'ingénieur.

- Cela peut être aussi dû au fait que le nombre d'instructions par secondes du processeur n'est pas une mesure précise. Le constructeur du processeur n'a en effet pas indiqué cette donnée, c'est pourquoi seul un benchmark permet de trouver une estimation du nombre d'instructions par seconde. Il s'agit d'une estimation réalisée à partir d'une moyenne lors d'un test entre différentes machines ayant le même processeur.

- Ensuite, il faut rappeler que d'autres processus peuvent "interférer" avec notre application. Ainsi, on ne peut pas être sûr que le processeur effectue uniquement des instructions de notre programme. Il peut, en même temps, faire des opérations pour d'autres processus.