Université de Montréal

Rapport du devoir #3 :
Planification d'un réseau de panneaux publicitaires électroniques

Par
Steven Han, 20129462
Yassine Hajouji Idrissi, 20038060
et
Chen Zong Jiang, 20122046

Département d'informatique et de recherche opérationnelle

Travail présenté à Samuel Ducharme
Dans le cadre du cours IFT-2015 section A
Structures de données

31 Juillet 2019

Préambule : Dans ce document, on dit souvent « temps de complexité » ou « ordre de grandeur ». Cela signifie la même chose.

1. Autoévaluation

1.1 Fonctionnement du programme

Notre programme fonctionne à 90%. Seulement le test du fichier carte3.txt n'est pas exactement pareil. On ne sait pas d'où vient ce problème, car on est sûr de notre algorithme (explication dans la section 1.2). Il y a sûrement un cas très particulier qui nous échappe. Nous n'allons pas expliquer le code en détail, vous pouvez voir le code source pour ça : on parlera seulement de l'algorithme qui mérite quelques soulignements. Notre fichier de sortie n'est pas exactement à la lettre comme les fichiers corrigés, mais si on inverse les sommets « fautifs », on a le même résultat.

Les sommets fautifs proviennent de l'incompréhension de l'énoncé. Selon le forum, on devrait imprimer selon le nom des rues d'entrés et non le chemin ARM. Par exemple, si dans un fichier carte0.txt on a : rue3 b c 4, alors la sortie devrait être rue3 b c 4 si on prend cette rue, même si le chemin de l'algorithme prend plutôt rue3 c b 4. Or, dans le fichier arm1.txt (rue18 BWI MIA 946 devrait être rue18 MIA BWI 946) et dans l'exemple de l'énoncé, cela contredit ce qui a été affirmer dans le forum. Qui croire, le forum ou l'énoncé? Nous avons codé bien avant les discussions sur le forum et donc, on a suivi l'énoncé.

Donc dans notre programme, on a imprimé selon le chemin de l'algorithme. Essentiellement, cela revient au même. Si le chemin passe par le sommet c puis le sommet b, il imprimera rue3 c b 4 et non rue3 b c 4.

1.2 Algorithme utilisé

Nous avons utilisé un algorithme de Prim-Jarnik pour traverser un graphe. Il y a trois variations de cet algorithme :

1. Utiliser une matrice d'adjacence $\left(O(|v|^2)\right)$
2. Utiliser un tas binaire et une liste d'adjacence $\left(O(|E|\log|V|)\right)$
3. Utiliser un tas fibonnaci et une liste d'adjacence $\left(O(|E|+|V|\log|V|)\right)$

V représente le nombre de sommet dans le graphe et E le nombre d'arrêts.

Le temps de complexité de l'algorithme varie selon le choix de l'implémentation, mais le 2 et le 3 sont presque similaires. Dans notre code, nous avons choisi le premier cas (matrice d'adjacence), car elle est plus simple à implémenter. Pour encore se faciliter la tâche, nos matrices et toutes manipulations de cet algorithme utilise des nombres entiers et non le nom des nœuds actuels. Chaque indice de la matrice correspondra à son nœud en ordre croissant

d'alphabet. Par exemple : si nos sommets/noeuds sont a, b, c, alors l'indice 0 correspondra au sommet a, l'indice 1 correspondra au sommet b, etc.

Pour les matrices d'adjacences, ce sera une matrice 2D où chaque élément aura le poids de l'arrêt entre les deux sommets correspondant aux indices de la matrice. Par exemple, si on a trois sommets a, b, c et que $a\ b\ 1, b\ c\ 2, c\ a\ 3$. Alors notre matrice aura cette forme : $\big[[0, 1, 3], [1, 0, 2], [3, 2, 0]\big]$.

Pour la priorité des nœuds, voici comment notre programme fonctionne :

1. Pour chaque nœud visité, on regarde, en ordre croissant de sommets, tous les arrêts qui ont le même poids que l'arrêt minimale (précédemment trouvé)
2. Dès qu'on tombe sur le premier, on le prend.

Cette façon de procédé s'assure que l'on prend toujours celui qui a le plus petit sommet de départ. En cas d'égalité de sommet de départ, on prend celui qui est le plus petit sommet d'arrivé. Mais même avec ce procédé, nous ne savons pas pourquoi le test 3 ne produit pas exactement la sortie voulue.

Il y a d'autres étapes de l'algorithme, mais on voulait seulement souligner l'essentielle pour ne pas vous mélangiez lorsque vous lirez le code source. Pour plus d'informations sur l'algorithme de Prim-Jarnik et son fonctionnement, vous pouvez aller voir ce lien[1]

[1] https://en.wikipedia.org/wiki/Prim%27s_algorithm

2. Analyse temporelle de l'algorithme

La seule partie du code qui mérite une analyse temporelle en profondeur est sur l'algorithme de Prim-Jarnik. Le reste n'est que des simples boucles ou des fonctions qui servent comme outils.

2.1 Analyse de la fonction sommetMin

Préambule : la variable n désignera le nombre de sommets du graphe et étant donné que nous avons choisi la matrice d'adjacence, le nombre d'arrêts n'influencera pas l'ordre de grandeur du code.

Voici le code pour cette fonction :

```java
public int sommetMin(int[] distance, Boolean[] lienSommets, int[][] matriceAdjacente){
    int min = Integer.MAX_VALUE;
    int indexMin = -10;

    // Boucle qui cherche l'arret minimum avec les sommets qu'on a visite
    for (int i = 0; i < nbSommets; i++){
        if (!lienSommets[i] && distance[i] < min){
            min = distance[i];
            indexMin = i;
        }
    }
    /* Priorise l'ordre alphanumerique des sommets (file d'attente prioritaire).
     * L'idee est que pour chaque sommet visite, on regardera leurs lignes dans la matrice d'adjacente. S'il y
     * a une egalite et que le sommet n'a pas encore ete visite, on le prend. On trouve toujours celui qui est
     * le plus petit.
     */
    Collections.sort(indexAChercher);
    for (Integer index : indexAChercher) {
        for (int j = 0; j < matriceAdjacente[index].length; j++) {
            if (matriceAdjacente[index][j] == min && !lienSommets[j]) {
                return j;
            }
        }
    }
    return indexMin;
}
```

Les deux premières déclarations sont instantanées. Leur temps de complexité sera alors $O(1)$.

Ensuite, il y a une boucle qui itère de 0 à $nbSommets$. $nbSommets$ peut être très grand et donc en pire cas la valeur sera n. Ce qui donne un temps de complexité de $O(n)$.

Puis, il y a un appel à la fonction $sort()$ de la classe $Collection$ de java. Cette fonction, selon la documentation[2], on peut lire ceci : « `This implementation is a stable, adaptive, iterative mergesort that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately`

[2] https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html

n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays.». Donc, le temps de complexité varie selon la liste à trier, mais en pire cas, ce sera $O(n/2)$.

Ensuite, il y a deux boucles imbriquées. Avec le même raisonnement dit précédemment pour la première boucle, on peut conclure que le temps de complexité est de $O(n) * O(n) = O(n^2)$. Les deux dépendent du nombre de sommets dans le graphe.

En conclusion, le temps de complexité pour cette fonction est l'addition de tous les temps énumérés, c'est-à-dire $O(1) + O(n) + O\left(\frac{n}{2}\right) + O(n^2) = O\left(1 + n + \frac{n}{2} + n^2\right) = O(n^2)$.

2.2 Analyse de la fonction primAlgo (la plus importante)

Voici le code de cette fonction :

```java
public ArrayList<String> primAlgo(int matriceAdjacente[][], ArrayList<String> listeDesNoeuds){
    ArrayList<String> sortieARM;
    // Les noeuds visites par notre parcours ARM dans l'ordre
    int[] noeudsVisites = new int[nbSommets];
    // Tableau qui contiendra tous les arrets minimal de chaque sommets
    int[] distances = new int[nbSommets];
    // Indique si un noeud a ete visite
    Boolean[] aEteVisitee = new Boolean[nbSommets];

    // Initialise nos tableaux
    for (int i = 0; i < nbSommets; i++){
        distances[i] = Integer.MAX_VALUE;
        aEteVisitee[i] = false;
    }

    // On commence au premier sommet
    distances[0] = 0;
    noeudsVisites[0] = -1; // Racine de ARM

    for (int i = 0; i < nbSommets - 1; i++){
        int min = sommetMin(distances, aEteVisitee, matriceAdjacente);
        aEteVisitee[min] = true;
        indexAChercher.add(min);
        for (int j = 0; j < nbSommets; j++){
            // On sait deja quel sommet chercher, il faut trouver quel element est le bon
            if (matriceAdjacente[min][j] != 0 && !aEteVisitee[j] && matriceAdjacente[min][j] < distances[j]){
                noeudsVisites[j] = min;
                distances[j] = matriceAdjacente[min][j];
            }
        }
    }
    sortieARM = sortieARM(noeudsVisites, matriceAdjacente, listeDesNoeuds);
    return sortieARM;
```

Toute déclarations au début de la fonction ne prend qu'une seule étape, donc $O(1)$.

Il y a ensuite une boucle qui va de 0 à $nbSommets$. En pire cas, on peut avoir un nombre de sommet très grand et par conséquent, le temps de cette boucle sera en pire cas $O(n)$. Tous les étapes dans cette boucle est instantané $\left(O(1)\right)$, donc l'ordre de grandeur sera $O(n)$.

Les deux déclarations suivantes sont des affectations d'éléments d'un tableau. Donc ce sera un temps de $O(1)$.

Ensuite, on voit qu'il y a deux boucles imbriquées où le premier va de 0 à $nbSommets - 1$. Cette première boucle itère en pire cas $n - 1$ fois. On voit que dans la première boucle, on fait appel à la fonction *sommetMin* qui a un ordre de grandeur de $O(n^2)$. Les autres opérations avant la deuxième boucle sont instantanées. La deuxième boucle itéra de 0 à $nbSommets$. Donc son temps de complexité sera de $O(n)$. Comme l'appel de la fonction *sommetMin* et la deuxième boucle sont imbriqués dans une boucle avec un ordre de grandeur de $O(n)$, le temps de complexité sera respectivement $O(n^3)$ et $O(n^2)$. Pour avoir l'ordre de grandeur de la boucle au complet il suffit les additionner, ce qui donne $O(n^3) + O(n^2) = O(n^3 + n^2) = O(n^3)$ (les plus grands exposant domine les plus faibles).

En conclusion, l'ordre de grandeur de cette fonction sera l'addition de tous les ordres de grandeur présent dans le code, c'est-à-dire

$$O(1) + O(1) + O(1) + O(1) + O(n) + O(1) + O(1) + O(n^3) = O(n^3)$$

Notre ordre de grandeur ne donne pas celle prévu $\left(O(n^2)\right)$. Ceci est à cause que l'on devait ajouter un ordre de priorité. Si on avait des arrêts égaux à plusieurs endroit, il fallait partir du premier sommet et aller en ordre croissant alphanumérique de sommets jusqu'à rencontrer l'arrêt le plus proche. Ce qui ajoutait une double boucle, car on avait une matrice 2D.

2.3 Analyse de la taille de l'entrée du fichier

Jusqu'à présent, nous avons beaucoup parlé du nombre de sommets du graphe. Il manque un autre détail à parler : la taille du fichier. Indirectement, on a parlé de tous ça, mais nous allons clarifier les choses. La première section des fichiers d'entrés (avant les premiers tirets) sont les sommets du graphe. Or, dans notre code, nous savons déjà que l'ordre de grandeur sera $O(n^3)$. La deuxième partie (entre les tirets) est en fait le nombre d'arrêt du graphe. Or, comme le nombre d'arrêt n'influence pas l'ordre de grandeur, l'ordre de grandeur restera toujours $O(n^3)$ peut importe le nombre de rues/arrêts dans la deuxième section. C'est la première section qui dicte le temps de notre algorithme.

3. Conclusion

Notre algorithme marche à 90% du temps : il y a quelques cas très particuliers qui nous échappe. Cependant, notre algorithme vérifie chaque nœud en ordre croissant alphanumérique et il est difficile à avaler le fait que notre algorithme ne marche pas toujours. Nous avons utilisé la matrice d'adjacence pour représenter l'algorithme de Prim-Jarnik et nous avons un ordre de grandeur de $O(n^3)$ (à cause des priorités) au lieu de $O(|n|^2)$.