IFT 2015 – Structures des donnée
Devoir #3

Manuela Girotti

1 Auto-évaluation

Le programme fonctionne correctement.

2 Code en Java

Voir le fichier `GIROTTI_Tp3.java` joint [1].

2.1 Structures des donnée utilisées

Pour le codage on a utilisé les structures suivantes :

— pour stocker les informations sur chaque arête, on crée une classe `Street` :

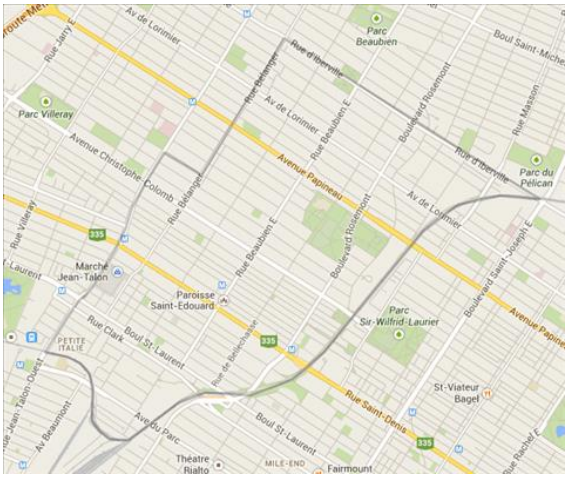
```
private static class Street{
        public String name;
        public String depart;
        public String arrive;
        public Double cost;

        public Street(String rue, String start, String end, Double c){
                name = rue;
                depart = start;
                arrive = end;
                cost = c;
                }
}
```

— pour le graph `citygraph` contenant la liste du sommets et des arêtes, on implémente un arbre binaire de recherche où chaque noeud est identifié par le nom du sommet (i.e. du site) et contient une liste chainée (la liste d'adjacence) avec noeuds de type `Street` :

```
TreeMap<String , LinkedList<Street>> citygraph
                = new TreeMap<String , LinkedList<Street >>();
```

Note 1. *On assume que le graphe du quartier est un graphe connexe, non-orientée et sans cycles.*



(a) La carte du quartier Rosemont-Petite Patrie à Montréal.

(b) La carte du centre-ville de Leuven (Belgique).

FIGURE 1 – Exemples des villes nord-américaines et européennes.

Note 2. *On assume que le graphe du quartier n'est pas trop dense ($|rues| \ll |sites|$), comme dans des villes nord-américaines, où la disposition des rues est régulière (*FIGURE *1a), et pas comme les villes européennes (*FIGURE *1b).*
Pour cette raison, l'implémentation du graph avec des listes d'adjacence prend moins de mémoire que l'implémentation avec une matrice d'adjacence.

— pour la liste des arêtes à visiter pendant l'exécution de l'algorithme de Prim, on implémente une queue prioritaire `PrimMST` où l'ordre est donnée par le coût d'installation dans chaque rue (pour ce fin, on implémente un comparateur `Q_comparator` *ad hoc*) :

```
PriorityQueue<Street> PrimMST = new PriorityQueue<Street>(Q_comparator);

Comparator<Street> Q_comparator = new Comparator<Street>() {
public int compare(Street str1, Street str2) {
        if (str1.cost != str2.cost) {
                return str1.cost.compareTo(str2.cost);
        }
        else {
                if (!str1.depart.equals(str2.depart)) {
                        return str1.depart.compareTo(str2.depart);
                }
                else {
                        return str1.arrive.compareTo(str2.arrive);
                }}
}};
```

— pour tenir compte des sommets visités pendant l'exécution de l'algorithme de Prim, on utilise un ensemble (`HashSet`) :

```
HashSet<String> visited_vertices = new HashSet<String>();
```

— l'arbre de recouvrement minimum est stocké dans un `TreeSet` où l'ordre est donnée par l'ordre lexicographique des sommets de départ (voir le comparateur `MST_comparator`) :

```
TreeSet<Street> MSTree = new TreeSet<Street>(MST_comparator);

Comparator<Street> MST_comparator = new Comparator<Street>() {
        public int compare(Street str1, Street str2) {
                if (!str1.depart.equals(str2.depart)) {
                        return str2.depart.compareTo(str1.depart);
                }
                else{
                        return str2.arrive.compareTo(str1.arrive);
                }
            }
};
```

(on aurait pu utiliser un `HashSet` ici, mais dans ce cas on aurait dû trier tous les éléments avant de les imprimer dans le fichier).

2.2 Pseudocode pour l'algorithme de Prim

L'algorithme de Prim en pseudo-code est le suivant [2] :

1. Initialiser une file de priorité.
2. Commencer avec un nœud arbitraire dans le graph (le site de base).
3. Insérer tous les arêtes connectées au noeud dans la file.
4. Bien que nous n'ayons pas encore visité tous les nœuds, supprimer la première arête (la moins chère) de la file. Ajoutez le nœud situé à l'extrémité opposée de cette arête à notre liste de nœuds visités. Insérer maintenant tous les autres arêtes de ce nœud dans la file.

Le code pour l'algorithme est le suivant :

```
private static TreeSet<Street> PrimJarnik
        (TreeMap<String,LinkedList<Street>> citygraph) {
        // Prim's algorithm

        Comparator<Street> MST_comparator = new Comparator<Street>() {
        public int compare(Street str1, Street str2) {
                if (!str1.depart.equals(str2.depart)) {
                        return str2.depart.compareTo(str1.depart);
                }
                else {
                        return str2.arrive.compareTo(str1.arrive);
                }
```

```
        }};

        Comparator<Street> Q_comparator = new Comparator<Street>() {
        public int compare(Street str1, Street str2) {
                if (str1.cost != str2.cost) {
                        return str1.cost.compareTo(str2.cost);
                }
                else {
                        if (!str1.depart.equals(str2.depart)) {
                                return str1.depart.compareTo(str2.depart);
                        }
                        else {
                                return str1.arrive.compareTo(str2.arrive);
                        }}
        }};


        HashSet<String> visited_vertices = new HashSet<String>();
        PriorityQueue<Street> PrimMST = new PriorityQueue<Street>(Q_comparator);
        TreeSet<Street> MSTree = new TreeSet<Street>(MST_comparator);

        String startV = citygraph.firstKey();
        visited_vertices.add(startV);
        citygraph.get(startV).forEach(item->PrimMST.add(item));

        while (visited_vertices.size() < citygraph.size()) {
                Street toinspect = PrimMST.poll();
                if (!visited_vertices.contains(toinspect.arrive)){
                        MSTree.add(toinspect);
                        visited_vertices.add(toinspect.arrive);
                        citygraph.get(toinspect.arrive)
                                .forEach(item->PrimMST.add(item));
                }
        }
        return MSTree;
}
```

3 Analyse temporelle théorique (pire cas)

Soit :
— N le nombre des sommets (sites) du graph ;
— M le nombre des arêtes (rues) du graph (rappel : $M \in \Omega(N) \cup \mathcal{O}(N^2)$).

On assume que toutes les opérations élémentaires (opérations algébriques, initialisation des variables, etc.) prennent un temps d'exécution constant $\mathcal{O}(1)$.

On assume aussi que chaque appel à `Scanner` et `BufferedWriter` (lecture et écriture d'un fichier) prends un temps d'exécution constant ($\mathcal{O}(1)$), puisque chaque ligne de texte dans les fichiers de input et output contiennent un nombre limité des caractères.

`parsing1` Pour chaque site, la méthode `addVertexToGraph` crée un nouveau nœud dans le graphe `citygraph` (`TreeMap`) avec clé le nom du site (sommet) et valeur une liste chainée vide :

$$\mathcal{O}(N).$$

`parsing2` Pour chaque arête du graphe lu par `Scanner`, la méthode `addEdgeToGraph` ajoute l'arête dans la liste d'adjacence du sommet de départ ($\mathcal{O}(\log N)$ pour le trouver dans l'arbre `citygraph` et $\mathcal{O}(1)$ pour l'ajout) et dans la liste du sommet d'arrivée (même complexité) :

$$\mathcal{O}\left(M \log N\right)$$

`écriture1` Écriture des N sommets de l'arbre de recouvrement minimum (et du graphe aussi) :

$$\mathcal{O}\left(N\right).$$

`PrimJarnik` La méthode `PrimJarnik` implémente l'algorithme de Prim. La complexité de chaque opération dans la méthode est indiquée ci-dessous :

- `firstKey()` dans le `TreeMap citygraph` prend $\mathcal{O}(\log N)$;
- `add` dans le `HashSet visited_vertices` prends $\mathcal{O}(1)$;
- `get()` dans le `TreeMap citygraph` prend $\mathcal{O}(\log N)$ pour chercher un sommet v et pour chaque arête qui départ du v on appelle la méthode `add` dans la `PriorityQueue PrimMST` qui prend $\mathcal{O}(\log M)$: en total

  $$\mathcal{O}(\log N + \deg(v) \log M)$$

  (où $\deg(v)$ est le degree du sommet v) ;
- le cycle `while` est répété au moins $N - 1$ fois et au plus $2M$ fois (toutes les arêtes sont sortées de la file `PrimMST`) :
  - `poll` dans la queue `PrimMST` prend $\mathcal{O}(\log M)$ (la queue `PrimMST` aura au plus $2M$ éléments) ;
  - `contains` et `add` dans le `HashSet visited_vertices` prennent $\mathcal{O}(1)$;
  - `add` dans le `TreeSet MSTree` prends $\mathcal{O}(\log M)$;
  - comme en avant, la ligne `citygraph.get(`*`vertex`*`).forEach(item->PrimMST.add(item))` prend

    $$\mathcal{O}(\log N + \deg(vertex) \log M)$$

    (cette opération est exécutée seulement si le sommet *vertex* n'a pas été encore visité).

En conclusion, la complexité de la méthode `PrimJarnik` est

$$\mathcal{O}(M \log N)$$

dans le pire cas, car $\sum_{v \in V} \deg(v) = 2M$ et $\log M \in \Theta(\log N)$.

Note 3. *La deuxième boucle* `while` *est purement esthétique. Sa complexité est* $\mathcal{O}(N \log N)$ *au pire cas.*

`écriture2` Calcul du coût total et écriture des $N-1$ arêtes de l'arbre de recouvrement minimum et du coût total :

$$\mathcal{O}(N).$$

La complexité totale du code est

$$\mathcal{O}(M \log N)$$

dans le pire cas.

A Code en Java

Plusieurs consultations de [1] pendant l'écriture du programme

```
import java.util.*;
import java.io.*;

public class Tp3 {

private static class Street{
        public String name;
        public String depart;
        public String arrive;
        public Double cost;

        public Street(String rue, String start, String end, Double c){
                name = rue;
                depart = start;
                arrive = end;
                cost = c;
        }
}


private static void addVertexToGraph
        (TreeMap<String,LinkedList<Street>> citygraph,String v) {

        LinkedList<Street> edges = new LinkedList<Street>();
        citygraph.put(v,edges);
}

private static void addEdgeToGraph
        (TreeMap<String,LinkedList<Street>> citygraph,
        Street newStreet) {
        // add the new edge to both the list of edges of the starting vertex AND
        // to the list of edges of the ending vertex
```

```
        citygraph.get(newStreet.depart).add(newStreet);

        Street newStreet2 = new Street(newStreet.name, newStreet.arrive,
                newStreet.depart,newStreet.cost);
        citygraph.get(newStreet.arrive).add(newStreet2);
}

private static TreeSet<Street> PrimJarnik
        (TreeMap<String,LinkedList<Street>> citygraph) {
        // Prim's algorithm

        Comparator<Street> MST_comparator = new Comparator<Street>() {
                public int compare(Street str1, Street str2) {
                        if (!str1.depart.equals(str2.depart)) {
                                return str2.depart.compareTo(str1.depart);
                        }
                        else {
                                return str2.arrive.compareTo(str1.arrive);
                        }
                }
        };

        Comparator<Street> Q_comparator = new Comparator<Street>() {
                public int compare(Street str1, Street str2) {
                        if (str1.cost != str2.cost) {
                                return str1.cost.compareTo(str2.cost);
                        }
                        else {
                                if (!str1.depart.equals(str2.depart)) {
                                        return str1.depart.compareTo(str2.depart);
                                }
                                else {
                                        return str1.arrive.compareTo(str2.arrive);
                                }}
                }
        };

        HashSet<String> visited_vertices = new HashSet<String>();
        PriorityQueue<Street> PrimMST = new PriorityQueue<Street>(Q_comparator);
        TreeSet<Street> MSTree = new TreeSet<Street>(MST_comparator);
        TreeSet<Street> MSTree_esthetic = new TreeSet<Street>(MST_comparator);

        String startV = citygraph.firstKey();
        visited_vertices.add(startV);
        citygraph.get(startV).forEach(item->PrimMST.add(item));
```

```
        while (visited_vertices.size() < citygraph.size()) {
                Street toinspect = PrimMST.poll();
                if (!visited_vertices.contains(toinspect.arrive)){
                        MSTree.add(toinspect);
                        visited_vertices.add(toinspect.arrive);
                        citygraph.get(toinspect.arrive)
                                .forEach(item->PrimMST.add(item));
                }
        }

        // extra few lines of code to comply with the request that
        // one needs to print the minimum spanning tree with first vertex and
        // second vertex in lexicographic order
        while (!MSTree.isEmpty()) {
                Street tocheck = MSTree.pollFirst();
                if (tocheck.depart.compareTo(tocheck.arrive)>0) {
                        Street updatestr =
                                new Street(tocheck.name,tocheck.arrive,
                                        tocheck.depart,tocheck.cost);
                        MSTree_esthetic.add(updatestr);
                }
                else {
                        MSTree_esthetic.add(tocheck);
                }
        }

        return MSTree_esthetic;
}


public static void main(String[] args) {

        File IN_fileName = new File(args[0]);
        String OUT_fileName = args[1];

        TreeMap<String,LinkedList<Street>> citygraph =
                new TreeMap<String,LinkedList<Street>>();

        try {
        // read the given file args[0]
        Scanner scan = new Scanner(IN_fileName);

        try {
        // write on the file named args[1]
```

```
FileWriter filewriter = new FileWriter(OUT_fileName);
BufferedWriter bufferedWriter = new BufferedWriter(filewriter);

while(scan.hasNextLine()) {
        // first part of the scanning process
        String token = scan.nextLine();
        char delimiter = '-';
        if (token.charAt(0) == delimiter) {
                break;
        }
        else {
                token = token.trim();
                addVertexToGraph(citygraph,token);
        }
}

while(scan.hasNextLine()) {
        // second part of the scanning process
        String token = scan.nextLine();
        char delimiter = '-';
        if (token.charAt(0) == delimiter) {
                break;
        }
        else {
                String[] split = token.split("[:,;]+");
                String newStreetName = split[0];
                String newStartPoint = split[1];
                String newEndPoint = split[2];
                String newCost_str = split[3];

                Double newCost = Double.parseDouble(newCost_str);
                Street newStreet
                        = new Street(newStreetName, newStartPoint,
                                newEndPoint, newCost);

                addEdgeToGraph(citygraph,newStreet);
        }
}
scan.close();


for (String label: citygraph.keySet()) {
        bufferedWriter.write(label);
        bufferedWriter.newLine();
}
```

```
        // *** Prim's algorithm ***
        TreeSet<Street> ARMtree = PrimJarnik(citygraph);

        double total_cost = 0;

        for (Street straat: ARMtree.descendingSet()) {
                bufferedWriter.write(String.format("%-10s",straat.name)
                        + String.format("%-8s",straat.depart)
                        + String.format("%-8s",straat.arrive)
                        + String.format("%-10.0f",straat.cost));
                total_cost += straat.cost;
                bufferedWriter.newLine();
        }

        bufferedWriter.write("---");
        bufferedWriter.newLine();
        bufferedWriter.write(String.format("%-10.0f",total_cost));

        bufferedWriter.close();

        }
        catch(IOException ex) {
        System.out.println("Error writing file '" + OUT_fileName + "'");}
        }
        catch(IOException ex) {
        System.out.println("Error reading file '" + IN_fileName + "'");}
        }
}
```

Références

[1] Java Platform, Standard Edition 8, API Specification : https://docs.oracle.com/en/.

[2] D. Brown, *Notes de cours CP164 – Data Structures I*, https://bohr.wlu.ca/cp164/, Wilfrid Laurier University.