Compte rendu AEA 2

Antoine Canda Claire Hunter

5 mai 2017

1 Introduction

Le but de ce TP est de mettre en pratique et d'expérimenter les différents algorithmes de calcul d'arbres recouvrants de coût minimum (*Minimum Spanning Tree* ou MST) avec notamment les implémentations des algorithmes de Prim et de Kruskal.

2 Implémentation des graphes

La première chose qu'il a été nécessaire de faire a été l'implémentation de nos graphes. Nous avons utilisé trois classes qui sont Vertex, Edge et GraphImpl qui implémente une interface Graph. Le graphe est composé d'une liste de sommets et une d'une liste d'arc ou arêtes.

Un sommet est tout simplement un identifiant qui est dans notre cas une chaîne de caractère représentant le nombre du sommet. Une arrête elle est composé d'un sommet source, d'un sommet cible ou destination et d'une valeur. Si le graphe n'est pas valué, la valeur par défaut des arêtes sera 1.

Les méthodes associées sont relativement simple comme obtention de la liste des sommets ou arêtes, l'ajout d'un sommet ou d'une arrête (valuée ou non), l'obtention d'un sommet par son nom et des arêtes reliés à un sommet.

3 Générateur de graphe d'Erdos-Renyi

Nous avons créé un générateur de graphes aléatoires afin de pouvoir expérimenter facilement nos algorithmes sur des graphes totalement aléatoires en faisant varier les paramètres que sont la densité du graphe (pourcentage de sommets reliés entre eux par une arête) et le nombre de sommets afin d'éprouver nos algorithmes dans différents cas.

Le générateur a besoin de deux informations capitales qui sont le nombre de sommets du graphe, N, et la probabilité que chaque sommet soit relié à un autre, p. Chaque sommet possédera donc environ $N * p$ arêtes.

Extrait de code du générateur :

```
/**
 * Class RandomGraphGenerator permettant de générer un graphe d'Erdos-Renyi de façon aléatoire
 * @author antoine et claire
 *
 */
public class RandomGraphGenerator {
/**
 * Générateur de graphe d'Erdos-Renyi
 * @param n le nombre de sommet du graphe
 * @param p la probabilité que deux sommets soient reliés par une arrête
 * @return un graph g
 * @throws VertexAlreadyExistsException
 * @throws VertexNotFoundException
```

```
 */
public static Graph generateErdosRenyiGraph(int n, float p) throws VertexAlreadyExistsException, V
// Initialisation du graphe et du générateur de nombre aléatoire.
Graph graph = new GraphImpl();
int i, j;
Random random = new Random();
double nbRandom;
int nbRandomN;
// Borne supérieure de la valeur d'une arrête
int N = (int) Math.pow(n, 4);

// Ajout des sommets au graphe
for(i = 0; i < n; i++) {
graph.addVertex(new Vertex(Integer.toString(i)));
}

// Ajout des arrêtes valuées au graphe
for(i = 0; i < n; i++) {
for(j = i+1; j < n; j++) {
// Calcul d'un flottant entre 0 et 1 dont le but est de déterminer si on ajoute une arrête entre l
nbRandom = random.nextFloat();

if(nbRandom <= p) {
// Calcul de la valeur de l'arrête
nbRandomN = random.nextInt(N) + 1;
graph.addEdge(graph.getVertexFromId(Integer.toString(i)), graph.getVertexFromId(Integer.toString(j
}
}
}

return graph;
}
}
```

Suite à la génération du graphe nous avons une méthode qui permet de le convertir en fichier avec la structure suivante et une autre permettant de créer un graphe à partir de ce fichier.

Une ligne correspond aux arêtes valuées d'un sommet et est composée d'une suite de nombres, chacun séparé par un espace. Le premier nombre correspond au sommet source auxquels sont reliés les différentes arêtes. Ensuite nous avons une succession de $2n$ nombres qui sont les couples (sommet destination / valeur de l'arête) associés au sommet source en question.

Exemple de fichier résultat :

```
1 2 7 3 10
2 3 2
3 4 21 5 9
4 5 19 7 6 8 5
5 6 8 9 3 8 13
6 10 20
8 9 1 11 4 12 11
9 10 12
```

4 Algorithme de calcul des arbres recouvrants de coût minimum

4.1 Algorithme de Prim

Nous avons implémenté une version naïve de l'algorithme de Prim qui ne sera pas efficace à cause des structures de données utilisées.

L'algorithme de Prim fait parti des algorithmes gloutons. Dans cette version, on récupère la liste de toutes les arêtes en utilisant une ArrayList et on va parcourir à chaque fois toute la liste pour en récupérer l'arête de poids minimal. A chaque fois on va tester si l'arête est adjacente à un sommet qui a déjà été visité. Le coût de parcours est en O(n) avec n la taille de la liste des arêtes. On effectue cette opération jusqu'à ce que le parcours de la liste ne donne plus d'arêtes dont seul un sommet est adjacent à un sommet visité (afin de pas produire de cycle). Une fois l'arête adjacente de poids minimum trouvée, on l'ajoute à la liste des arêtes résultantes et on la retire de la liste des arêtes non traitées (ce qui se fait en temps O(n) une fois de plus.

Nous avons essayé d'améliorer cet algorithme en utilisant une structure optimisée pour récupérer l'arête de poids minimum à savoir un tas de Fibonacci. Une version initiale a été faite et améliore considérablement le temps mais n'est pas complètement optimisée pour autant. Cette version (totalement optimisée) n'est pas fonctionnelle pour le rendu et a été retiré.

Je suppose que le problème dans l'implémentation vient de la gestion des arrêtes dans le tas de Fibonacci, avec notamment l'utilisation de decreaseKey().

Le principal avantage est que toutes les opérations qui se font sur cette structure sont en temps amorti (on compense les quelques fois où l'opération coûte cher par la plupart des cas où elle est bon marché en calculant la moyenne du coût de l'opération sur une suite d'opérations) soit en O(1) pour l'insertion, l'accès au minimum, l'union et la diminution de clé, soit en O(log n) pour les autres qui sont la suppression et la suppression du minimum. En comparaison, la liste chaînée n'a que l'insertion et l'union en temps constant ; toutes les autres opérations se font en O(n).

On en déduit que plus le graphe sera gros, plus on gagnera en efficacité.
Extrait de code :

```
/**
 * Version légèrement optimisée de l'algorithme de Prim utilisant un tas de Fibonacci
 * @param graph
 * @return
 */
public static CoupleResultat primOpti(GraphImpl graph) {
// System.out.println("Debut de l'algorithme de Prim");
long startingTime = System.nanoTime();

List<Edge> res = new ArrayList<>();

List<Vertex> vertices = graph.getVertices();

Vertex vertex = vertices.remove(0);

// On obtient la liste des arrêtes du graphe que l'on mélange
// aleatoirement
List<Edge> edges = graph.getEdgesFromVertex(vertex);
FibonacciHeap fh = new FibonacciHeap();

for (Edge edge : edges) {
fh.insert(edge, edge.getValue());
}
// On cree la liste des sommets visites et on ajoute le sommet source de
```

```
// la premiere arrete de la liste
List<Vertex> vertexVisited = new ArrayList<Vertex>();

// On recupere la premiere arete de poids min et on ajoute sommet source
// et destination à l'ensemble
Edge edge = (Edge)fh.removeMin();

res.add(edge);
vertexVisited.add(vertex);
Vertex newVertex = (edge.getSource().equals(vertex))? edge.getDest() : edge.getSource();
vertexVisited.add(newVertex);

for (Edge e : graph.getEdgesFromVertex(newVertex)) {
fh.insert(e, e.getValue());
}

while (!fh.isEmpty()) {


Edge edgeMin;

// On recupere l'arrête de poids min
//

edgeMin = (Edge)fh.removeMin();

// On verifie si le sommet source ou cible est present dans
// l'ensemble visite pour trouver une arrete adjacente a ces
// derniers
boolean sommetSourcePresent, sommetDestinationPresent;
sommetSourcePresent = vertexVisited.contains(edgeMin.getSource());
sommetDestinationPresent = vertexVisited.contains(edgeMin.getDest());

// On ajoute arrête et sommet nouvellement visité aux ensembles
if ((sommetSourcePresent || sommetDestinationPresent)
&& (!sommetSourcePresent || !sommetDestinationPresent)) {
res.add(edgeMin);
Vertex vertexCourantNonVisite = (sommetSourcePresent) ? edgeMin.getDest() : edgeMin.getSource();
vertexVisited.add(vertexCourantNonVisite);

for (Edge e : graph.getEdgesFromVertex(vertexCourantNonVisite)) {
fh.insert(e, e.getValue());
}
}
}

long endingTime = System.nanoTime();
//System.out.println( "Fin de l'algorithme de Prim. Temps d'execution : " + (endingTime - starting
CoupleResultat cpl = new CoupleResultat(endingTime - startingTime, res);
return cpl;
}
```

4.2 Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme qui permet de construire un arbre recouvrant minimum. En pratique, on commence par trier les arêtes par poids croissant. Un algorithme de tri optimisé l'effectue avec une complexité en O(n log n) avec n étant le nombre d'arêtes comme par exemple le tri fusion. Ensuite on regarde pour chaque arête si elle appartient à une composante

connexe acyclique dont le coût est minimal. Les opérations de base de cet algorithme sont la création d'ensembles connexes, la fusion des composantes connexes et la vérification qu'il n'y a pas de cycle. Une structure de type Union-Find est optimal pour cet algorithme.

Le code de notre algorithme Kruskal :

```
/**
 *
 * Classe implémentant l'algorithme de Kruskal
 * @author antoine et claire
 *
 */
public class Kruskal {

/**
 * Fonction permettant de savoir dans quel numero de composante connexe dans la liste se trouve ur
 * @param idVertex L'identifiant du sommet a verifier
 * @param compConnexes La liste des composantes connexes de l'arbre couvrant a l'etat actuel
 * @return Le numero de la composante connexe trouvee
 */
public static int compConnexe(String idVertex, ArrayList<ArrayList<Edge>> compConnexes) {
boolean found = false;
Iterator<ArrayList<Edge>> it = compConnexes.iterator();
ArrayList<Edge> comp = null;
Iterator<Edge> itEdge;
Edge edge;
int numComp = -1;

// On cherche dans chaque composante connexe tant qu'on n'a pas trouve
while(it.hasNext() && !found) {
comp = (ArrayList<Edge>)it.next();
numComp++;
itEdge = comp.iterator();
// On parcourt toutes les aretes de cette composante connexe jusqu'a
// les avoir toutes parcourues ou avoir trouve une arete qui comporte
// le sommet que l'on cherche.
while(itEdge.hasNext() && !found) {
edge = itEdge.next();
if(edge.getDest().getId().equals(idVertex) || edge.getSource().getId().equals(idVertex))
found = true;
}
}

if(!found)
return -1;
else
return numComp;
}

/**
 * Methode de fusion des composantes connexes de l'arbre couvrant a l'etat actuel
 * @param numC1 Le numero de la premiere composante connexe a fusionner
 * @param numC2 Le numero de la deuxieme composante connexe a fusionner
 * @param compConnexes La liste des composantes connexes de l'arbre couvrant a l'etat actuel
 */
public static void mergeCompConnexe(int numC1, int numC2, ArrayList<ArrayList<Edge>> compConnexes)
compConnexes.get(numC1).addAll(compConnexes.get(numC2));
// Ne pas oublier de retirer la deuxieme composante connexe
// maintenant qu'elle est fusionnee avec la premiere
compConnexes.remove(numC2);
```

```
}

/**
 * Fonction qui verifie la presence d'un cycle ou non dans l'arbre couvrant a l'etat actuel si l'o
 * @param idsVerticesVisited L'ensemble des identifiants des sommets deja visites
 * @param edge L'arete qu'on veut ajouter a l'arbre couvrant si possible
 * @param compConnexes La liste des composantes connexes de l'arbre couvrant a l'etat actuel
 * @return Booleen a true si un cycle est present, false sinon
 * @throws Exception
 */
public static boolean checkCycle(Set<String> idsVerticesVisited, Edge edge, ArrayList<ArrayList<Ed
String eDestId = edge.getDest().getId(),
eSrcId = edge.getSource().getId();
// Comme on travaille avec un Set, si en essayant d'ajouter
// l'identifiant du sommet au Set on retourne false de la fonction,
// alors c'est que cet identifiant de sommet y etait deja present.
boolean destAdded = idsVerticesVisited.add(eDestId),
sourceAdded = idsVerticesVisited.add(eSrcId);
int numC1, numC2;

// Si aucun des identifiants des deux sommets de l'arete n'etaient presents
// dans la Set, alors c'est que cette arete n'est pas encore liee a l'arbre.
// C'est donc une nouvelle composante connexe.
if(sourceAdded && destAdded) {
ArrayList<Edge> newCompConnexe = new ArrayList<>();
newCompConnexe.add(edge);
compConnexes.add(newCompConnexe);
}

// Sinon, si soit l'identifiant de la source de l'arete, soit l'identifiant de la
// destination de l'arete etaient deja presents, alors il faut chercher la composante
// connexe ou se trouve cet identifiant et y ajouter l'arete
else if(sourceAdded && !destAdded) {
numC1 = compConnexe(eDestId, compConnexes);
if(numC1 == -1)
throw new Exception("Sommet d'identifiant " + eDestId + " non trouve dans une composante connexe !
compConnexes.get(numC1).add(edge);
} else if(!sourceAdded && destAdded) {
numC2 = compConnexe(eSrcId, compConnexes);
if(numC2 == -1)
throw new Exception("Sommet d'identifiant " + eSrcId + " non trouve dans une composante connexe !"
compConnexes.get(numC2).add(edge);
}

// Sinon, si les deux identifiants des deux sommets de l'arete etaient deja presents,
// alors il faut verifier une condition
else { // if(!sourceAdded && !destAdded)
numC1 = compConnexe(eDestId, compConnexes);
numC2 = compConnexe(eSrcId, compConnexes);
if(numC1 == -1)
throw new Exception("Sommet d'identifiant " + eDestId + " non trouve dans une composante connexe !
else if(numC2 == -1)
throw new Exception("Sommet d'identifiant " + eSrcId + " non trouve dans une composante connexe !"
// Si les deux sommets sont dans la meme composante connexe, alors il y a un cycle !
if(numC1 == numC2)
return true;
// Sinon, l'arete est une liaison entre les deux composantes connexes (l'une contenant le sommet
// source et l'autre contenant le sommet destination). Il faut alors fusionner ces deux composante
```

```
// connexes en une seule.
else
mergeCompConnexe(numC1, numC2, compConnexes);
}
return false;
}

/**
 * Fonction permettant de trouver l'arbre couvrant de poids minimal dans un graphe passe en parame
 * @param graph Le graphe a considerer
 * @return La liste des aretes faisant partie de l'arbre couvrant de poids minimal
 */
public static List<Edge> kruskal(GraphImpl graph) {
System.out.println("Debut de l'algorithme de Kruskal");
long startingTime = System.nanoTime();

List<Edge> res = new ArrayList<>();
Set<String> idsVerticesVisited = new TreeSet<>();
ArrayList<ArrayList<Edge>> composantesConnexes = new ArrayList<>();
graph.getEdges().sort(null);
Iterator<Edge> it = graph.getEdges().iterator();
Edge edge;

while(it.hasNext() &&
( idsVerticesVisited.size() != graph.getVertices().size()
|| ( (idsVerticesVisited.size() == graph.getVertices().size()) && composantesConnexes.size() != 1)
) {
edge = it.next();
try {
if(!checkCycle(idsVerticesVisited, edge, composantesConnexes)) {
res.add(edge);
}
} catch (Exception e) {
e.printStackTrace();
return null;
}
}

long endingTime = System.nanoTime();
System.out.println("Fin de l'algorithme de Kruskal. Temps d'execution : "+ (endingTime-startingTim
return res;
}
}
```

5 Résultat de l'expérimentation

Différentes expérimentations ont été faites afin d'évaluer les performances. Les algorithmes ont été validés sur plusieurs graphes de tailles relativement faible afin de pouvoir valider le résultat en déroulant l'algorithme à la main et en comparant les résultats trouvés (c'est de cette façon qu'on a pu déterminer une erreur dans le prim optimisé) et ensuite on a fait des tests de performances notamment sur le temps en demandant le temps système au début de l'algorithme et en demandant le même temps à la fin et en faisant la soustraction.

Pour valider un temps moyen, on a calculé une moyenne en fonction de n valant 100, 1000 et 5000 et p variant de 0.1 à 0.9 par pas de 0.2 sur une cinquantaine de graphe. L'algorithme de Prim n'a été testé que pour n valant 100 car dès qu'on arrive à 1000 arêtes on obtient un temps de 5313 secondes ce qui n'est absolument pas surprenant vu l'algorithme.

Tableau des résultats :

Algorithme	n	p = 0.1	p = 0.3	p = 0.5	p = 0.7	p = 0.9
Kruskal	100	0.002269786	0.00173356	0.002074426	0.002536112	0.003179448
Prim	100	0.045633083	0.11698075	0.218843847	0.321000451	0.412442426
PriOpti	100	0.007251972	0.01550704	0.023559879	0.031538982	0.039853049
Kruskal	1000	0.158169541	0.250489992	0.369454977	0.449094961	0.552147824
PrimOpti	1000	4.408913919	14.617865728	18.815617298	33.983118388	44.005737823
Kruskal	5000	5.160898551	12.294321235	15.01691839	20.382204407	23.558814084

En analysant les résultats, on voit directement la différence entre une version optimisé et une version qui ne l'est pas. Sur un graphe avec 1000 sommets, avec kruskal on met 0.45s quand on a une densité de 70% des sommets reliés entre eux contre plus de 5000s pour le même graphe avec Prim. Le rapport est de plus de 10 000. Pour des graphes de taille bien plus modeste (100 sommets) on est déjà 20 à 100 fois plus rapide en général.

La différence entre Prim et PrimOpti est relativement faible pour 100 sommets (un facteur de 10 environ) alors que pour 1000 sommets on est plutôt de l'ordre de 100. Mais on retrouve la même différence entre PrimOpti et Kruskal.

Il est intéressant de relever l'évolution des temps de calcul en fonction de la densité des sommets reliés entre eux. En effet l'évolution n'est pas linéaire, si on regarde pour 1000 sommets reliés à 10% ou à 90%, le rapport en temps de calcul est inférieur à 4 mais le nombre d'arêtes lui a augmenté de façon importante dans le même temps.

Pour 5000 sommets, les résultats sont là plus intéressants, car le temps nécessaire augmente cette fois de manière plus importante qu'avant notamment entre une densité très faible et une densité plus importante (0.1 à 0.3) alors qu'entre 0.3 et 0.5 l'augmentation a été plus contenue (3s contre 7s). Le temps nécessaire a été multiplié par 3 entre une densité de 0.1 et 0.5 et de moins de 5 entre 0.1 et 0.9. Si on compare avec 1000 on a des résultats environ 50 fois plus longs pour un nombre de sommets 5 fois plus importants.

6 Conclusion

A travers ce TP, on a pu tester les deux algorithmes de Kruskal et Prim et avoir une vraie réflexion sur l'importance du choix des structures de données que l'on utilise et l'impact que ces derniers peuvent avoir sur le temps d'exécution. Dans nos implémentations, on a un algorithme de Kruskal qui est plutôt efficace et qui supporte une montée à l'échelle en conservant des temps de calcul largement acceptable alors que l'algorithme de Prim (naïf) ne peut tout simplement pas le faire faute d'optimisation. En revanche une version optimisée de l'algorithme de Prim pourrait probablement s'approchait de l'efficacité de la version de Kruskal car la partie la plus coûteuse de l'algorithme est l'obtention de l'arête de poids minimum qui est la force du tas de Fibonacci mais il nous a manqué un peu de temps pour corriger les erreurs dans l'implémentation.