IR2 - Algorithmique des graphes

Fiche 5 - Algorithme de Dijkstra, algorithme de Kruskal

L'objectif de ce TP est d'implanter l'algorithme de Dijkstra. La structure de donnée à choisir est celle de **liste d'adjacence**. Le langage de programmation est le Java.

Étant donné un graphe non orienté pondéré par des réels positifs ou nuls, l'algorithme de Dijkstra permet de calculer le chaîne de poids minimale entre deux sommets s et t [1]. Le poids d'une chaîne est la somme des poids de chacune de ses arêtes qui la constituent.

Exercice 1. Algorithme de Dijkstra
L'algorithme de Dijkstra procède de la manière suivante. Pour chaque sommet, on enregistre sa plus courte distance par rapport à une source s. Au début, les distances sont initialisées à l'infini pour tous les sommets autres que s et sont mises à jour étape par étape. Une étape consiste à sélectionner parmi les sommets non visités le sommet x qui possède la plus petite distance et à mettre à jour les distances des sommets y qui lui sont adjacents en testant si le passage par x pour atteindre y minimise la distance pour atteindre y depuis s. L'algorithme s'arrête lorsque tous les sommets ont été visités. Cet algorithme nécessite :

- un tableau d qui contient pour chaque sommet sa distance par rapport à s ;
- une liste *visite* qui contient les sommets qui n'ont pas encore étés totalement visités ;
- un tableau *parent* qui contient pour chaque sommet son sommet parent, c'est à dire le sommet adjacent qui permet de l'atteindre dans la chaîne de poids minimale. Ce tableau est utilisé pour reconstituer la chaîne de poids minimale de s à t.

Voici l'algorithme :

1. Pour les graphes avec des poids négatifs, il faut utiliser l'algorithme de Ford-Bellman.

Algorithm 1 DIJKSTRA(G, s, t)

Require: un graphe G non orienté pondéré par réels positifs ou nuls, s et t deux sommets de G.
Ensure: retourne le poids de la chaîne de poids minimale passant par s et t.

```
1: visite ← ListeVide()
2: for tout sommet x de G do
3:    parent(x) ← null
4:    d(x) ← ∞
5:    AjouterDansListe(visite, x)
6: end for
7: d(s) ← 0
8: while visite n'est pas vide do
9:    x ← sommet de la liste visite avec d(x) minimal.
10:   SupprimerDansListe(visite, x)
11:   for tout sommet y adjacent à x do
12:      if d(y) > d(x) + poids({x, y}) then
13:         d(y) ← d(x) + poids({x, y})
14:         parent(y) ← x
15:      end if
16:   end for
17: end while
18: afficher la chaîne de poids minimale à l'aide du tableau parent.
19: return d(t)
```

1. Implanter l'algorithme de Dijkstra.
2. Évaluer sa complexité temporelle et spatiale.
3. Une implantation directe de la recherche du sommet non visité de plus petite distance produit une complexité temporelle en $\Theta(n)$ où n est le nombre de sommets du graphe. Est-il possible faire mieux ? Si oui, expliquer la manière de faire.

Étant donné un graphe non orienté pondéré par des réels, l'algorithme de Kruskal permet de trouver un arbre couvrant de poids minimum, c'est à dire un ensemble d'arêtes qui ne forment pas de cycle, tel que tout sommet du graphe appartienne à au moins une arête et que la somme des poids des arêtes soit le plus petit possible. L'algorithme de Dijkstra permet de répondre à ce problème.

Exercice 2. Algorithme de Kruskal

1. Implanter une méthode d'union-find pour gérer des partitions disjointes d'ensembles de sommets.
2. Utiliser la méthode d'union-find pour implanter l'algorithme de Kruskal.
3. Évaluer sa complexité temporelle et spatiale.