Exercice : Problème de plus court chemin dans un graphe

1 Présentation du problème

Un graphe pondéré est un graphe dont les arêtes possèdent chacune un poids. Ce poids peut représenter par exemple une distance, un temps de parcours ou un coût d'une autre nature pour traverser cette arête (un payage par exemple).

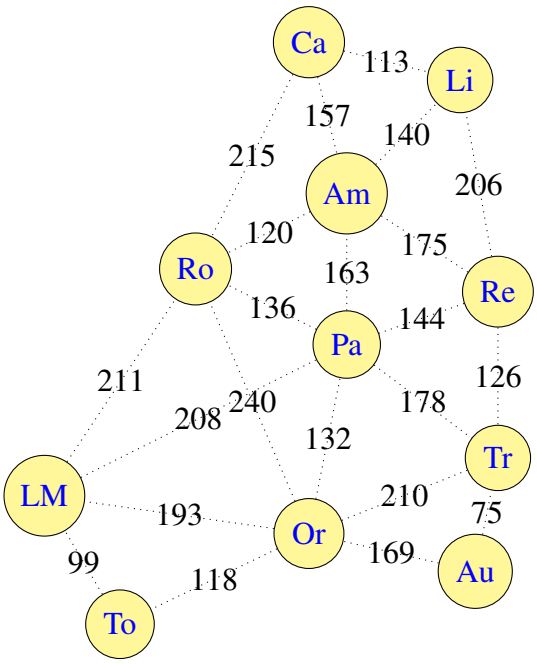

FIGURE 1 – Représentation de villes sous forme d'un graphe. Ici les poids sont les distances en km entre les villes. Ce graphe est contenu dans le fichier *villes.txt*.

Le problème que l'on souhaite résoudre est le suivant : étant donné un sommet de départ et un sommet d'arrivée, trouver le plus court chemin entre ces deux

sommets. On pourra se déplacer dans le graphe en traversant une ou plusieurs arêtes successives. On calculera la longueur d'un chemin en sommant les poids des arêtes traversées. On considérera que le graphe n'est pas orienté (ce n'est pas nécessaire) et qu'il est connexe (ce n'est pas nécessaire).

2 Algorithme de Dijkstra

Le graphe considéré contient N sommets (ici villes). On va stocker dans un tableau *dist* les distances de tous les sommets du graphe au sommet de départ. Ce tableau sera donc de taille N. De même, on va stocker dans un tableau *ant* d'entiers de taille N l'antécédent associé à chaque sommet (le sommet par lequel il faut passer pour suivre le chemin optimal). On va également utiliser une *priority_queue* : une liste d'arêtes qui contiendra les arêtes que l'on doit étudier prochainement. Détaillons maintenant la façon dont on va remplir ces éléments.

Au départ, on considère que toutes les distances sont de -1 (dans *dist*) (sauf pour le sommet de départ qui a une distance zéro), tous les antécédents sont -1 et la *priority_queue* contient les arêtes autour du sommet de départ.

Tant que la *priority_queue* (PQ) n'est pas vide, on prend l'arête de plus petit poids dans cette PQ (obtenue avec l'instruction *top()*). On prend l'extrémité de fin de cette arête (celle qui n'a pas encore été étudiée). Si ce sommet a déjà été visité, alors un chemin plus court a déjà été trouvé vers ce sommet, aucune modification ne doit être apportée aux listes et on peut passer à l'arête suivante de la PQ. Sinon, on définit l'antécédent du sommet visité comme le sommet précédent. De plus, on ajoute dans PQ les arcs vers les sommets voisins avec un poids qui sera défini comme la distance (à l'origine) du sommet déjà visité à laquelle on ajoute le poids de l'arc utilisé pour passer au sommet suivant. Bien entendu on parcourt la PQ en prenant à chaque fois l'arête de plus petit poids (on obtient cette arête avec l'instruction *top()*, pour supprimer cette arête de la PQ et donc passer à l'arête suivante, on utilise *pop()*).

Une fois que la PQ est vide, alors chaque sommet a sa distance au sommet de départ (en prenant le plus court chemin). Et la liste des antécédent contient la route qu'il faut utiliser pour suivre le plus court chemin du sommet de départ vers le sommet d'arrivée.

Algorithm 1 Algorithme de Dijkstra

1. On initialise une *priority_queue* PQ avec la liste des arêtes partant du sommet de départ. On initialise également la liste d'antécédents avec -1 pour tous les noeuds et la liste des distances avec -1 pour tous les nœuds sauf le nœud de départ où on met 0.

for j dans PQ **do**

2. On récupère les extrémités a et b de j.

3. Pour chacun de ces sommets s'il n'a pas encore été visité on effectue les opérations suivantes (sinon on passe à l'arc suivant de PQ ; nb : au moins un des deux sommets a deja été visité) :

4. On stocke l'antécédent de ce sommet et la distance au sommet de départ (qui est donnée par le poids de l'arc j).

5. On ajoute à PQ les arcs voisins de ce sommet avec un poids modifié de la façon suivante : on prend la distance du noeud courant au sommet de départ et on y ajoute le poids de l'arc considéré.

end for

Une fois que PQ est vide, tous les sommets ont une distance et un antécédent (sauf le sommet de départ).

6. On prend alors le sommet d'arrivée et on déroule les antécédents jusqu'au sommet de départ.

3 Outils utiles

Priority queue Une *priority queue* est une file à laquelle on ajoute une priorité de passage : un élément à haute priorité apparaîtra avant un élément à basse priorité. On utilisera la classe *priority_queue* <*Arc*> de la bibliothèque standard.

— *push* permet d'ajouter un élément
— *top* récupère l'élément de plus haute priorité
— *pop* enlève l'élément de plus haute priorité
— *empty* renvoie un *boolean* indiquant si la file est vide

Pour utiliser cette classe il faut donner une relation d'ordre à *Arc* qui définira le niveau de priorité dans la file.

4 Instructions

1. Charger les fichiers sources de départ et le Makefile ; les compiler.

2. On veut utiliser une *priority_queue<Arc>*. Pour cela, il faut définir une relation d'ordre sur les *Arc*. Définir cette relation d'ordre de manière à ce que l'*Arc* le plus grand soit celui qui a le plus petit poids.
3. Construire une *priority_queue<Arc>*. Vérifier que l'instruction *top()* renvoie l'*Arc* de plus petit poids.
4. Passons maintenant à la fonction *GPS*. **Initialisation.** Les vecteurs *ant* et *dist* ont déjà été initialisés. Initialiser la *priority_queue<Arc> PQ*. Il faudra la remplir avec la liste des *Arc* partant du point de départ.
5. **Boucle principale.** Coder une boucle *while* qui s'arrête quand *PQ* est vide. A chaque itération de la boucle, on récupère l'*Arc* de plus petit poids et on considère a et b ses extrémités. Si a et b ont déjà été visités (valeur ≥ 0 dans *dist*), on passe tout de suite à l'itération de boucle suivante. Si l'un des deux n'a pas encore été visité, alors on définit son antécent comme étant l'autre sommet et sa distance à l'origine comme étant le poids de l'*Arc* courant. De plus, on ajoute à *PQ* tous les autres *Arc* partant de cette extrémité avec un poids modifié comme étant la distance de l'extrémité au départ plus le poids de l'*Arc*.
6. **Calcul du chemin optimal.** Une fois que l'on sort de la boucle *while*, le tableau *ant* contient pour chaque sommet le sommet d'où il faut venir pour suivre le chemin optimal. A partir de ce tableau, reconstituer le chemin optimal. La distance alors parcourue est contenue dans le tableau *dist*. Renvoyer un message dans le terminal indiquant le chemin optimal et la distance à parcourir.
7. **Tests.** Essayer de calculer plusieurs trajets à partir de la fonction *GPS*. Vérifier que l'itinéraire proposé est optimal.