Feuille d'exercices n°6 de l'option d'informatique.

Thème : Recherche d'un plus court chemin dans un graphe.

Objectifs

On considère $G = (S, A)$ un graphe orienté caractérisé par la liste de ses sommets S et la liste de ses arcs $A \subset S \times S$ ainsi que par une application $\ell : A \longrightarrow \mathbb{R}^+$ qui, à chaque arc associe sa longueur. On définit la longueur d'un chemin $(s_0, s_1, \ldots, s_p)$ comme la somme des longueurs des arcs qui la composent :

$$\sum_{1 \leqslant i \leqslant p} \ell(s_{i-1}, s_i)$$

Étant donné deux sommets s et t, on se propose de déterminer la longueur minimale $\ell^*(s, t)$ d'un chemin menant de s à t (s'il existe un tel chemin).
L'algorithme de Dijkstra est basé sur le résultat suivant qui est une simple conséquence de l'inégalité triangulaire :
Lemme : si $(s_0 = s, s_1, \ldots, s_p = t)$ est un chemin de longueur minimale menant de s à t, alors $(s_0, \ldots, s_i)$ est un chemin de longueur minimale menant de s à s_i, quel que soit $i \in [\![0, p]\!]$.

Mise en place

L'algorithme consiste à répartir les sommets en trois catégories :

i) les sommets u connus, c'est à dire ceux pour lesquels on connaît $\ell^*(s, u)$,

ii) les sommets u en attente, c'est à dire, les sommets qui ne sont pas connus et qui sont adjacents à un sommet connu,

iii) les sommets u qui n'appartiennent à aucune des 2 catégories précédentes.

On construit de proche en proche une liste $C_k = (s_0, s_1, \ldots, s_k)$ de sommets connus, avec $s_0 = s$, et telle que pour tout $i \in [\![0, k]\!]$, et tout $u \notin C_{i-1}$, ou bien $\ell^*(s, s_i)$ est majoré par $\ell^*(s, u)$ ou bien il n'existe pas de chemin menant de s à u.
À chaque sommet u en attente, on associe $\ell_k(u)$, longueur minimale d'un chemin menant de s à u et ne passant que par des sommets connus, à l'exception bien entendu du dernier; l'indice k s'explique par le fait qu'au fur et à mesure que l'on ajoute des sommets connus, on peut diminuer cette longueur minimale. On note V_k l'état de la liste des sommets en attente, au moment où l'on s'apprête à déterminer s_{k+1}. À chaque sommet u est donc associé la valeur $\ell_k(u)$.

- Initialisation : $C_0 = (s_0 = s)$, $\ell^*(s_0) = 0$ et V_0 contient les sommets adjacents à s.
- Itération : on aborde l'étape k. Si $s_k = t$, l'algorithme a réussi : $\ell^*(s_k)$ est la longueur minimale d'un chemin menant de s à t. Sinon, on pose $V_{k+1} = V_k \cup \{\text{sommets adjacents de } t\}$ et $\ell_{k+1}(u)$ est défini pour $u \in V_{k+1}$ par:

$$\ell_{k+1}(u) = \begin{cases} = \ell_k(u) \text{ si } u \text{ non adjacent à } s_k \\ = \min(\ell_k(u), \ell_k(s_k) + \ell(t, s_k)) \text{ si } u \text{ adjacent à } s_k \text{ et } u \in V_k \\ = \ell_k(s_k) + \ell(t, s_k) \text{ si } u \text{ adjacent à } s_k \text{ et } u \notin V_k \end{cases}$$

  On dispose maintenant de V_{k+1}; si cette liste est vide, l'algorithme s'arrête, traduisant le fait qu'il n'existe pas de chemin menant de s à t. Sinon, soit u un élément de V_{k+1} tel que $\ell_{k+1}(u)$ soit minimal; alors $\ell^*(u) = \ell_{k+1}(u)$. En effet, s'il existait un chemin menant de s à u, de longueur inférieure strictement à $\ell_{k+1}(u)$, ce chemin devrait passer par un certain nombre de sommets connus, avant de passer par un premier sommet v en attente (le passage par un sommet inconnu

étant clairement impossible), exhibant ainsi un chemin menant de s à v, de longueur strictement inférieure à $\ell_{k+1}(u)$, et contredisant donc l'hypothèse de minimalité.

On pose alors $s_{k+1} = u$, $\ell^*(s_{k+1}) = \ell_{k+1}(u)$, et on enlève u de la liste des sommets en attente. Ceci achève la k-ième itération; il est assez facile de voir que les hypothèses faites à l'entrée dans cette étape se retrouvent *mutatis mutandis* à l'entrée de l'étape suivante.

Notons que l'algorithme donne un résultat correct, même pour le cas particulier $s = t$.

Remarque concernant la programmation
L'algorithme précédent conduit à la structure de programme suivante :

```
1 (* initialisation *)
2 tant que la liste des sommets en attente est non vide
  et que le dernier sommet connu est différent de t faire
3      - recherche du sommet en attente s' le plus proche de s
4      - ajouter s' à la liste des sommets connus
5      - supprimer s' de la liste des sommets en attente
6      - ajouter les sommets adjacents à s' qui ne sont pas dans la
         liste des sommets connus
7 fin tant que
```

Remarques complémentaires
Il est clair que le fait de travailler avec un graphe non orienté n'apporte pas de simplification significative; d'ailleurs certaines applications de l'algorithme requièrent que le graphe soit orienté: que l'on pense par exemple aux durées des trajets en avion, qui ne sont pas nécessairement les mêmes dans les sens aller et retour.

Si t n'est pas dans la même composante connexe que s, on obtient à la fin de l'algorithme la liste des sommets qui peuvent être atteints en partant de s, *id est*, dans le cas d'un graphe non orienté: sa composante connexe.

Il est facile d'obtenir, comme sous-produit de l'algorithme, la liste des sommets formant un chemin de longueur minimale menant de s à t: lorsque l'on ajoute un sommet u dans la liste des sommets en attente, on note que son prédécesseur doit (en l'absence d'autre information) être s_k; lorsque l'on trouve un chemin plus court pour un sommet en attente, on met à jour cette indication; enfin, lorsqu'un sommet en attente devient connu, on enregistre définitivement son prédécesseur. On prouve que ceci fournit effectivement le chemin demandé, en utilisant à nouveau le lemme énoncé au début du texte.

Vous trouverez définis dans le fichier `carte.ml` le type `ville` qui regroupe les villes placées sur notre carte et A qui contient la liste des arcs de notre graphe c'est à dire ici les couples de villes «adjacentes» avec la distance reliant ces deux villes sous la forme suivante :

```
type ville = Paris | Lyon | Marseille | Rennes | Bordeaux;;
let A = [((Paris,Lyon),350);((Lyon,Marseille),400)];;
```

Exemple : À partir du fichier `carte.ml`, on obtient le résultat suivant pour relier Lille à Nice.

```
# chemin_min A Lille Nice;;
- : int * ville list =
 1191, [Lille; Paris; Dijon; Macon; Lyon; Avignon; Aix; Nice]
```

Note bibliographique et historique
Edsger Wijbe Dijkstra a présenté cet algorithme en 1959 dans un article de la revue *Numerische Mathematik*.