Programmation en C et structures de données

guillaume.revy@univ-perp.fr

CC2, Vendredi 16/12/2022 de 13h à 18h

Ce programme est à réaliser seul, sous environnement GNU/Linux. Il doit être rendu sur Moodle, sous forme d'un unique fichier `<nom_prenom.c>`. Seuls les documents disponibles sur Moodle sont autorisés. Tout autre document (cours, exercices, ...), ou autre matériel (téléphone portable, ...) est interdit.

Conseils :
- Il est à noter que la qualité primera sur la quantité : un programme court, clair, bien conçu et qui compile sera mieux noté qu'un gros programme mal conçu, incompréhensible, voire qui ne compile pas. Utilisez des noms de variables explicites, indentez votre programme, et n'hésitez pas à le commenter.
- Lisez tout le sujet avant de commencer.
- Veuillez respecter les noms de fonctions et structures proposées dans le sujet.

Un GPS est une application qui permet, étant données une ville de départ et une ville d'arrivée, de déterminer le plus court chemin entre ces deux villes. Pour ce faire, l'application GPS utilise une représentation de la carte routière sous forme d'un graphe.

De manière simplifiée, un graphe est une structure constituée d'un ensemble de **sommets** reliés entre eux par des **arcs** ou **arêtes**, chaque arc étant étiqueté par un **poids**. Dans le cas du GPS, les sommets sont les villes, les arcs sont les routes reliant deux villes, et le poids de chaque arc indique la distance entre les villes reliées par cet arc. Nous considérons ici des graphes non orientés, c'est-à-dire, pour lesquels les arcs sont bi-directionnels (si il y a un arc entre deux sommets s_1 et s_2, cela signifie qu'il y a un lien de s_1 vers s_2 et un de s_2 vers s_1, et qu'ils sont de même poids). Ceci est illustré sur la Figure 1, où les villes sont représentées par des numéros, et où aller de 1 vers 2 et de 2 vers 1 coûte 3.

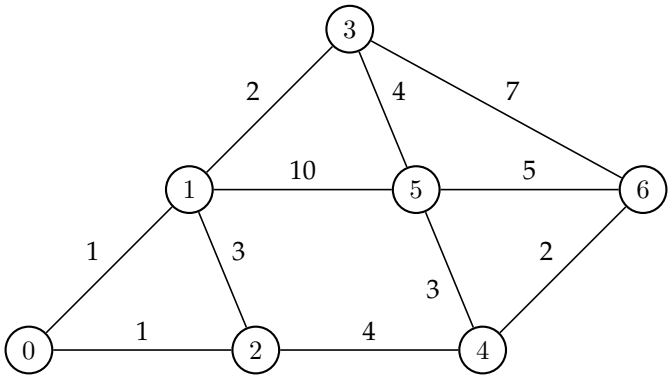

FIGURE 1 – Exemple de graphe.

L'objectif de ce TP est d'écrire un programme qui permet de déterminer le plus court chemin entre deux villes, c'est-à-dire, entre deux sommets d'un graphe stocké dans un fichier, en utilisant l'algorithme de Dijkstra.

1. Représentation d'une liste de couples (sommet, valeur)

Nous allons dans un premier définir une **liste simplement chaînée de couples (sommet, valeur)**. Cette structure nous servira pour la suite de l'exercice.

► **1.** Écrire une structure `couple`, qui permet de représenter un couple (sommet, valeur), où le sommet et la valeur sont deux entiers positifs ou nuls.

► **2.** Écrire une structure `liste_couples`, qui permet de représenter une liste simplement chaînée de couples (sommet, valeur).

Enfin sur cette structure de données, nous allons définir cinq fonctions.

► **3.** Écrire la fonction `inserer_couple` qui, étant donnés une liste chaînée de couples $\mathcal{L}$, un sommet s et une valeur d , insère le couple (s, d) en queue dans la liste $\mathcal{L}$.

► **4.** Écrire la fonction `extraire_couple_min_valeur` qui, étant donnée une liste $\mathcal{L}$, extrait de $\mathcal{L}$, en le supprimant, et retourne un des couples (par exemple, le premier) qui a la plus petite valeur.

► **5.** Écrire la fonction `mettre_a_jour_couple` qui, étant donnés une liste chaînée de couples $\mathcal{L}$, un sommet s et une valeur d_2, permet de mettre à jour la valeur des couples (s, d) de la liste $\mathcal{L}$ en la remplaçant par d_2, ou d'insérer le couple (s, d_2) en queue de $\mathcal{L}$ si aucun couple (s, d) n'existe.

► **6.** Écrire la fonction `afficher_couples` qui permet d'afficher le contenu d'une liste de couples passée en paramètre.

► **7.** Écrire enfin la fonction `detruire_couples` qui permet de libérer la mémoire allouée pour une liste $\mathcal{L}$ passée en paramètre.

Vous pourrez bien évidemment créer un programme principal pour tester ces fonctions.

2. Représentation d'un graphe

Maintenant, nous allons voir comment représenter un graphe. Soit $\mathcal{G}$ un graphe de n nœuds, tous étiquetés de 0 à $n-1$, chaque nœud représentant une ville. Nous allons représenter un graphe par un **tableau de noms** et une **liste d'adjacence**. Plus particulièrement, pour chaque sommet $s \in \mathbf{N}$, le graphe stocke le nom de la ville correspondant et une liste d'arcs, c'est-à-dire, une liste de couples (d, ℓ) où d est un sommet du graphe et ℓ la longueur de l'arc reliant le sommet s au sommet d. Par exemple, sur le graphe de la Figure 1 (où $n = 7$), nous aurons

$$\mathcal{G}.\text{arcs}[2] = \{(0, 1), (1, 3), (4, 4)\}.$$

Et cette liste peut donc être manipulée via la liste simplement chaînée que l'on a créée à l'exercice précédent, où la valeur sera la longueur.

► **1.** Écrire une structure `graphe`, qui permet de représenter un graphe.

► **2.** Écrire une fonction `construire_graphe` qui construit et retourne un graphe stocké dans un fichier passé en paramètre

Le format de fichier utilisé est le suivant :

— le nombre n de sommets dans le graphe,
— le nom des n villes,
— une liste d'arcs sous la forme `s1:s2:d`, indiquant un arc *bi-directionnel* entre les sommets s_1 et s_2 de longueur d.

Des fichiers exemples sont disponibles sur Moodle. Le fichier correspondant au graphe de la Figure 1 est donné ci-dessous.

```
7           --> nombre de sommets dans le graphe
ville1      \
ville2       |
ville3       |
ville4        > liste du nom des 7 villes
ville5       |
ville6       |
ville7      /
0:1:1       \
0:2:1        |
1:2:3        |
1:3:2        |
1:5:10       |
2:4:4         > definition des argcs entre les villes
3:5:4        |
3:6:7        |
4:5:3        |
4:6:2        |
5:6:5       /
```

► **3.** Écrire une fonction `trouver_sommet` qui, étant donnée une chaîne de caractères représentant un nom de ville, retourne l'indice du nœud correspondant dans le graphe, et -1 si cette ville n'existe pas.

► **4.** Écrire une fonction `afficher_graphe` qui permet d'afficher le graphe, notamment, pour chaque nœud, la ville et la liste d'arcs. Pour le graphe de la Figure 1, l'affichage pourra être le suivant.

```
G[0] : ville1 -> (1, 1) (2, 1)
G[1] : ville2 -> (0, 1) (2, 3) (3, 2) (5, 10)
G[2] : ville3 -> (0, 1) (1, 3) (4, 4)
G[3] : ville4 -> (1, 2) (5, 4) (6, 7)
G[4] : ville5 -> (2, 4) (5, 3) (6, 2)
G[5] : ville6 -> (1, 10) (3, 4) (4, 3) (6, 5)
G[6] : ville7 -> (3, 7) (4, 2) (5, 5)
```

► **5.** Écrire enfin une fonction `detruire_graphe` qui permet de libérer la mémoire allouée pour un graphe $\mathcal{G}$ passé en paramètre.

3. Algorithme de Dijkstra

Pour implanter l'algorithme de Dijkstra, nous allons utiliser une **file de priorité**. Une telle file est en réalité une liste d'éléments avec chacun une priorité. *Et cette liste peut encore être manipulée via la liste simplement chaînée que l'on a créée à l'exercice précédent, où la valeur sera la priorité.*

L'algorithme de Dijkstra est donné ci-dessous. Il utilise les variables suivantes :

— un tableau `visites` de taille n, initialisé à 0 et où `visites[s]` indique le nombre de fois où l'algorithme à visiter le sommet s,

— un tableau `distance_min` de taille n, initialisé à 0 et où `distance_min[s]` indique la distance minimal courante entre le sommet de départ et le sommet s,

— et une file de priorité `file_priorite` initialisée avec une liste vide.

```
dijkstra(G, depart, arrivee)
  // entree: depart, arrivee = deux entiers positifs ou nuls
  // sortie: plus courte distance pour aller de depart a arrivee
  visites = {0, ... 0}
  distance_min = {0,...,0}
  file_priorite = {}

  ajouter le couple (depart, 0) a file_priorite
  distance_min[depart] = 0
  tant que file_priorite n'est pas vide faire
    (s, d) = extraire de file_priorite le couple avec la plus petite priorite
    visites[s] = visites[s] + 1
    si s == arrivee alors
      quitter la boucle
    fsi
    si s pas deja visite alors
      pour chaque arc (suc, len) sortant de s faire
        distance = distance_min[s] + len
        si distance_lin[s] == 0 ou distance < distance_min[s] alors
          distance_min[s] = distance
          mettre a jour ou inserer (suc, distance) dans file_priorite
        fpour
      fpour
    fsi
  ftant
  retourner distance_min[arrivee]
```

► **1.** Écrire une fonction `dijkstra` qui, étant donnés un graphe $\mathcal{G}$, l'indice d'un sommet de départ s_1 et celui de l'arrivée s_2, détermine et retourne la longueur du plus court chemin entre s_1 et s_2 dans $\mathcal{G}$.

► **2.** Écrire un programme principal qui prend le nom de la ville de départ et de celle d'arrivée en ligne de commande, et qui détermine et affiche la plus courte distance qui relie ces deux villes.

► **3.** Proposer une solution pour retourner également le plus court chemin, c'est-à-dire, l'ensemble des sommets traversés.