Cours de programmation séquentielle

Les graphes - algorithmes de plus cours chemins

1 Préambule

Ce travail pratique est organisé en plusieurs parties traitant la notion de recherche de plus court chemin dans un graphe. Il faudra implémenter l'algorithme de Dijkstra que vous avez vus en cours. Vous appliquerez ces algorithmes sur un graphe représentant un certain nombre de lignes ferroviaires suisses. Finalement, une partie graphique vous sera demandée pour rendre le travail un peu plus ludique.

Pour cette première partie vous serez assez libres dans le design de votre code. Des bouts de code vous sont fournis, vous devrez les adapter à vos besoin que cela soit pour le contenu des fonctions ou leurs signatures.

2 Les deux représentations d'un graphe

Dans ce travail, nous ne considérons que des graphes pondérés, non-orientés. Cela signifie que les sommets du graphe sont reliés par des arêtes et chacune d'entre elle porte un poids. Contrairement aux graphes orientés, les arête n'ont pas de direction (voir la fig. 1). Les poids seront des **entiers positifs** dans ce travail.

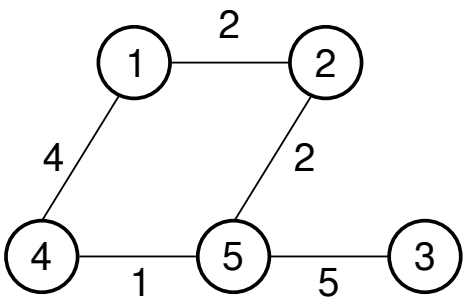

Figure 1: Exemple de graphe non-orienté.

Un graphe a deux représentations principales:

1. La liste d'adjacence.
2. La matrice d'adjacence.

Nous allons très brièvement décrire ces deux structures de données sur l'exemple de la fig. 1.

2.1 La liste d'adjacence

Pour un graphe non-orienté, la liste d'adjacence, est la liste des voisins pour chaque sommet. La longueur de cette liste est donc le nombre de sommets. Pour le cas de la fig. 1, elle a une longueur de 5.

Pour chaque sommet, nous avons ensuite une liste chaînée contenant l'identifiant du voisin, ainsi que le poids de l'arête y menant. L'ordre dans lesquels sont placés les sommets voisins n'a pas d'importance.

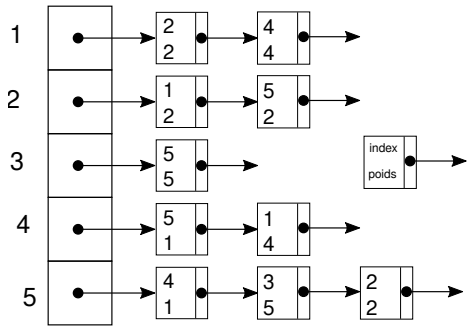

Figure 2: Liste d'adjacence du graphe de la fig. 1.

Sur la fig. 2 nous pouvons voir la liste d'adjacence correspondant au graphe de la fig. 1. Cette structure sera représentée par un tableau de liste chaînée dans ce projet.

2.2 La matrice d'adjacence

La matrice d'adjacence d'un graphe avec N sommets est de taille $N \times N$ et contient l'information sur la connectivité du graphe, ainsi que les différentes pondérations des arêtes. Dans le cas de la fig. 1, nous avons donc une matrice, $\underline{\underline{A}}$, de taille 5×5.

$$\underline{\underline{A}} = \begin{pmatrix} \infty & 2 & \infty & 4 & \infty \\ 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 5 \\ 4 & \infty & \infty & \infty & 1 \\ \infty & \infty & 5 & 1 & \infty \end{pmatrix}, \tag{1}$$

où l'élément A_{ij} de la matrice représente le poids de l'arête reliant le sommet i au sommet j. Les poids infinis signifient qu'il n'y a pas d'arête entre les sommets.

De façon très surprenante la structure de données qui sera utilisée pour la matrice d'adjacence dans ce travail est une ... matrice d'entiers.

3 Lecture de fichier XML

Dans un premier temps, vous devez construire les deux représentations des graphes discutées ci-dessus à partir du fichier XML, villes.xml, que vous trouverez sur Cyberlearn également. Pour ce faire, il faut utiliser la librairie `libxml2` et les squelettes de code xml_parser.h et xml_parser.c. Il s'agit ici de compléter ce code afin de faire deux choses:

1. Créer un tableau contenant les noms des villes, ainsi que leurs positions sur la carte.
2. A partir du tableau des villes et des informations de la connectivité, créer la liste et la matrice d'adjacence. Les poids des arêtes représentent le temps de parcours entre les villes.

Puis, vous avez également un squelette de main_dijkstra.c contenant les formats d'entrées/sorties pour pouvoir comparer avec les tests, ainsi qu'un Makefile.

3.1 Structures de données

Vous devez donc créer différentes structures pour contenir les informations décrites ci-dessus. Tout d'abord la structure contenant les informations sur les villes dans la structure `city`

```
#define NAME_LEN 50
typedef struct _city {
  char name[NAME_LEN];
  int longitude;
  int latitude;
} city;
```

Pour stocker un tableau de villes, utilisez un tableau dynamique de type `city`. Vous avez deux autres choix si vous le souhaitez. Un tableau dynamique dont la taille s'adapte en fonction du nombre d'éléments stockés (ce type est appelé communément `vector` ou mot dérivé dans différents langages). Vous trouverez un énoncé pour implémenter ce genre de structure sur ce lien. Autrement, vous pouvez également utiliser un vecteur "générique" si vous l'avez déjà implémenté ou si vous souhaitez le faire (mais ce n'est pas obligatoire). Mais cela nécessite la manipulation de `void *` qui sont au-delà de ce que nous avons vu en cours, mais que cela ne vous en empêche pas. Vous avez un énoncé de travail pratique se trouvant sur ce lien.

Puis pour la liste d'adjacence, vous devez créer un tableau de listes chaînées de type `connection`

```
typedef struct _connection {
    int neighbor;
    int weight;
    struct _connection* next;
} connection;
```

Finalement, pour la matrice d'adjacence inspirez-vous de la structure de matrices que vous avez créée au premier semestre. Il vous suffit de remplacer le type contenu dans la matrice d'un double en un entier signé.

4 Quand vous avez terminé

Quand vous avez terminé le travail ci-dessus, attaquez-vous à l'algorithme de Dijkstra que vous avez vu au cours d'algorithmique. N'oubliez pas de vous servir de la file de priorité.

5 Tests

5.1 Exécuter le programme et rediriger les sorties standard et d'erreur dans un fichier

`./monprog input.txt 1>std_output 2>err_output`

Exécute le programme `monprog` avec l'argument `input.txt` et redirige la sortie standard dans le fichier `std_output` et la sortie d'erreur dans le fichier `err_output`.

5.2 Comparer deux fichiers avec diff

`diff` est une commande qui permet de comparer deux fichiers, elle affiche le numéro des lignes et les lignes qui diffèrent entre deux fichiers et n'affiche rien si deux fichiers sont identiques.

`diff -Z file1 file2`

Compare deux fichiers sans tenir comptes des espaces à la fin des lignes. Si la sortie de votre programme est juste, la comparaison entre la sortie fournie et la sortie de votre programme ne devrait rien afficher.

On vous fournit un fichier de tests complet avec ce travail pratique. En effet, le fichier cmd_a_tester_dijkstra.txt est le fichier contenant les entrées à fournir à votre programme et output_dijkstra sont les sorties attendues. Ainsi en comparant les sorties de votre programme quand vous lui fournissez `cmd_a_tester_dijkstra.txt` en argument devrait donner ce qui est contenu dans `output_dijkstra`.

6 Remarques

1. Une fois le tableau des villes créé, vous devez faire une correspondance entre le nom de la ville et un indice. Pour ce faire vous pouvez simplement utiliser l'indice de la ville dans le "tableau des villes".
2. Pour représenter les poids infinis, utiliser la macro `INT_MAX` se trouvant dans `limits.h`.
3. Si elle n'est pas encore installée sur votre système, vous devez installer la librairie `libxml2`. Si vous utilisez une distribution Ubuntu ou basée sur ArchLinux, faites les commandes suivantes:

   - Ubuntu

     ```
     sudo apt install libxml2 libxml2-dev
     ```

   - Manjaro/ArchLinux

     ```
     sudo pacman -Sy libxml2
     ```

   Pour compiler avec les fichiers sources et obtenir le code objet vous devez utiliser l'option `-I` qui permet d'inclure un chemin spécifique.

```
gcc -c xml_parser.c -I/usr/include/libxml2
gcc -c main.c -I/usr/include/libxml2
```

Pour linker il faut ensuite utiliser l'option `-lxml2`.

4. Les exemples pour l'utilisation de la lecture de fichiers XML compilent[1], mais il manque des arguments aux fonctions. Il faut les rajouter pour que votre code remplisse les différentes structures de données nécessaires au bon déroulement de ce travail. Notez également que ces fonctions sont récursives.

5. N'essayez pas de faire passer tous les tests d'un coup. Il faut faire ça de façon incrémentale.

[1] Ils génèrent le code objet avec la commande `gcc -c xml_parser.c -I/usr/include/libxml2`.