Projet : Plus Court Chemin dans un graphe

Notions : Graphes

1 Introduction

Ce projet sera réalisé en binôme, sur les 5 dernières séances d'informatique ET en dehors des séances d'informatique. Pour la première séance, vous devez avoir lu le sujet et formé les binômes.

Attention : la réussite de ce projet exige du travail en dehors des séances.

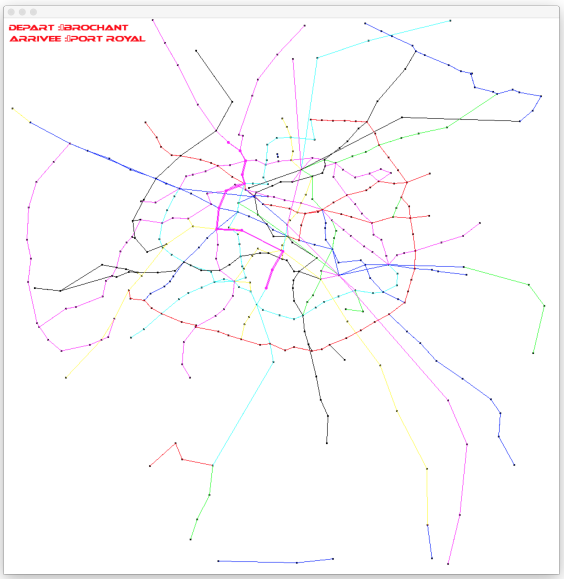

FIGURE 1 – Réseau et meilleur itinéraire (en gras et mauve) entre les stations Brochant et PortRoyal.

Le premier objectif du projet est de comparer 2 algorithmes calculant le meilleur itinéraire, i.e. le plus court chemin entre 2 villes ou 2 stations de métro parisien. La notion de plus court s'applique à la somme des coûts des arcs que suit notre chemin. Ce coût peut être n'importe quelle quantité, i.e. une distance, un temps de parcours, etc.

Le deuxième objectif du projet est de confronter vos codes à des données de taille relativement importante. Les réseaux routiers que vous allez utiliser comportent de 200 000 à 50 000 000 données. S'il est facile de traiter des problèmes de petite taille, il est plus difficile de traiter ceux de grande taille, qui peuvent mettre en évidence plus facilement des erreurs ou des insuffisances du code.

Le troisième objectif est de travailler correctement en équipe, même si le terme équipe est exagéré pour 2 membres. Une bonne organisation, communication écrite et verbale sont indispensables entre les 2 membres de l'équipe, ainsi qu'une certaine maîtrise des outils de développement `(gcc, gdb ou lldb, valgrind)` et de partage du code `(git)`.

Le réseau routier (ou le métro), se représente facilement par un graphe, chaque sommet étant une intersection de routes (une station de métro). Les arcs représentent les routes (les lignes de métro). Le coût d'un arc est la distance ou le temps entre 2 intersections (stations). Il existe de nombreuses méthodes de calcul d'un plus court chemin dans un graphe, plus ou moins performantes en fonction du problème, de la structure du graphe et des connaissances a priori.

2 Graphes

2.1 Terminologie et définitions

Graphe = couple $\mathcal{G}(X, A)$ où X est un ensemble de nœuds ou sommets (*vertices*) et A est l'ensemble des paires de sommets reliés entre eux (arêtes du graphe ou « arc »).

Arc (*edge*)= arête orientée.

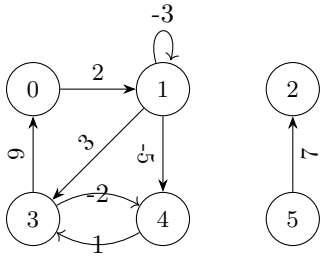

FIGURE 2 – Un exemple de graphe.

Chemin (*path*) = séquence d'arcs menant d'un sommet i à un sommet j.

Circuit (*circuit* ou *cycle*)= chemin dont les sommets de départ et d'arrivée sont identiques.

Valuation, coût (*cost*) = valeur numérique associée à un arc ou à un sommet.

Degré d'un sommet (*degree*) = nombre d'arêtes ayant ce sommet pour extrémité.

Voisins (*neighbours*) : les voisins d'un sommet sont les sommets qui lui sont reliés par un arc.

2.2 Représentation des sommets et des arcs

Pour nos calculs d'itinéraires :

— un sommet sera représenté par une structure contenant son nom, sa latitude et sa longitude (pour dessiner le graphe), la ligne de métro à laquelle ce sommet appartient dans le cas du métro et la **liste de ses successeurs**. C'est cette liste de successeurs (ou de voisins) qui permet de facilement représenter l'ensemble des données contenues dans un graphe.

— un arc sera une structure contenant le numéro du sommet d'arrivée et le coût de l'arc.

— la liste des successeurs d'un sommet est une liste chaînée des arcs qui partent de ce sommet. A partir d'un sommet, on peut donc facilement accéder à ses sommets voisins en parcourant cette liste, et uniquement à ses voisins. On utilisera une liste chaînée dynamique, comme vu en cours.

Les types de données ressembleront à ce qui suit. Il vous faudra les modifier pour ajouter des informations qui vous seront utiles. Par exemple, il sera pertinent d'ajouter un champ `cout` aux sommets.

```
// Type arc
typedef struct {
   int arrivee;     // L'indice du sommet d'arrivée de l'arc
   double cout;     // Le cout (distance) de l'arc
} edge_t;

// Type liste chaînée d'arcs
typedef struct maillon_edge {
   edge_t val;          // ou edge_t* val, suivant votre choix : liste d'arcs ou de pointeurs d'arcs
   struct maillon_edge * next;
}* listedge_t;

// Type sommet
typedef struct {
    int numero;         // indice du sommet
    char* nom;          // nom donné au sommet
    char* ligne;        // nom de la ligne, utile uniquement pour le métro
    double x,y;         // coordonnées latitude et longitude du sommet
    listedge_t edges;   // liste des arcs qui partent de ce sommet
    double pcc;         // valeur du "plus court chemin" entre le sommet de départ et ce sommet.
                        // n'est utile que durant le déroulé de l'algorithme.
} vertex_t;
```

2.3 Représentation du graphe

Un graphe se représente par une structure contenant un tableau de sommets (voir figure 3) et les nombres d'arcs et de sommets. Chaque sommet contient les informations définies ci dessus, auxquelles on pourra ajouter d'autres éléments utiles aux algorithmes décrits dans les parties suivantes, si nécessaire.

```
// Type graphe :
typedef struct {
  int size_vertices;    // nombre de sommets
  int size_egdes;     // nombre d'arcs
  vertex_t* data;     // tableau des sommets, alloué dynamiquement
} graph_t;
```

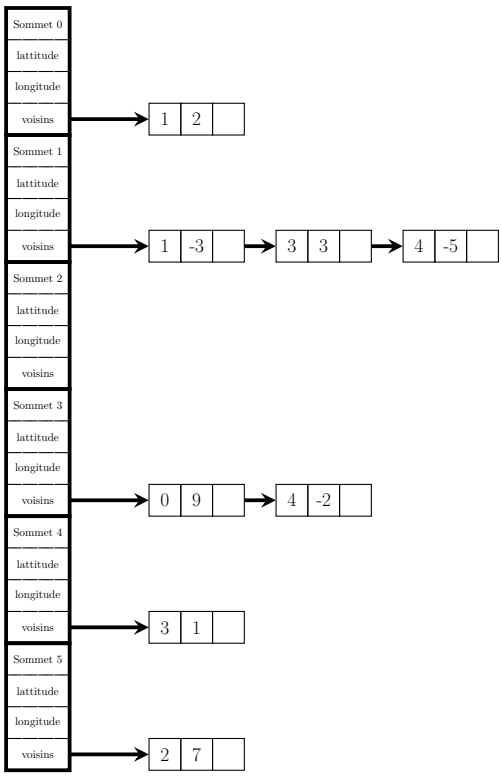

FIGURE 3 – Représentation du graphe de la figure 2 par listes chaînées de successeurs.

3 Plus court chemin dans un graphe

Dans un problème de plus court chemin , on considère un graphe orienté $\mathcal{G} = (X, A)$. Chaque arc $a_i \in A$ est muni d'un coût p_i. Un chemin $C =< a_1, a_2, ..., a_n >$ possède un coût qui est la somme des coûts des arcs qui constituent ce chemin. Le plus court chemin d'un sommet d à un sommet a est le chemin de coût minimum qui va de d à a. Il existe plusieurs algorithmes de calcul du plus court chemin :

- l'algorithme de *parcours en profondeur d'abord* ou *Depth Fisrt Search (DFS)*, examine les noeuds de manière similaire au parcours dans un arbre. On examine le noeud de départ, on choisit un voisin de ce noeud, puis un voisin du voisin, puis un voisin du voisin du voisin, etc... Cette stratégie peut être améliorée en choisissant le meilleur voisin selon un critère qui dépend du problème traité.
- l'algorithme de *parcours en largeur d'abord* ou *Breadth Fisrt Search (BFS)*, examine les noeuds de manière similaire au parcours en largeur dans un arbre. On examine le noeud de départ, puis tous les voisins de ce noeud, puis tous les voisins des voisins, etc... Cette stratégie peut être améliorée en ordonnant les voisins selon un critère qui dépend du problème traité.
- l'algorithme de *Bellman-Ford* est le seul à pouvoir s'appliquer dans le cas d'arcs à valeur négative. Attention cependant aux circuits de valeur négative, car il n'existe pas de solution dans ce cas.
- l'algorithme de *Dijkstra*, dans le cas d'arcs à valuation positive, est celui utilisé dans le routage des réseaux IP/OSPF, sans doute le plus populaire.
- l'algorithme $A*$ ou *Astar* utilise une heuristique pour trouver rapidement une solution approchée, mais peut aussi donner la solution exacte. Il ne s'applique qu'aux graphes dont les arcs sont à valuation positive. Il nécessite de disposer d'une fonction heuristique pour estimer le coût a priori entre deux nœuds. Il est souvent utilisé dans les problèmes de type labyrinthe ou itinéraire.

Le problème que nous nous proposons de traiter consiste à chercher le **meilleur parcours entre un sommet de départ et un sommet d'arrivée**, sur un graphe ne comportant que des valuations positives. Ce graphe est assez *sparse*, chaque sommet n'ayant que peu de voisins. Nous allons comparer les algorithmes de **Dijkstra** et **Astar**.

3.1 Algorithme de Dijkstra

L'algorithme de Dijkstra peut se voir comme la construction de tous les chemins partant du point de départ, selon des distances croissantes, jusqu'à trouver le sommet d'arrivée. On manipule deux ensembles :

Atraiter : contient les sommets qui ont déjà été atteints au moins une fois par un chemin depuis *depart*.

Atteint : contient les sommets qui ont été définitivement atteints, pour lesquels on est sûr qu'il n'existe pas d'autre meilleur chemin.

Algorithm 1 Int Dijkstra(int depart, int arrivee, graph_t graphe)

Entrées depart, l'indice du sommet de depart
Entrées arrivee, l'indice du sommet d'arrivee
Entrées graphe, le graphe
1: Initialiser un ensemble de sommets *Atraiter*, vide
2: Initialiser tous les *pcc* et les *Couts* des sommets à ∞
3: Mettre le *pcc* et le *Cout* du sommet *depart* à 0
4: Ajouter le sommet *depart* a l'ensemble *Atraiter*
5: **tant que** l'ensemble *Atraiter* n'est pas vide **faire**
6: Rechercher et supprimer le sommet u de plus faible $Cout$ de l'ensemble *Atraiter*
7: **si** u==*arrivee* **alors**
8: on a trouve la solution, retourner 1 et fin de la fonction
9: **else**
10: {On a trouve le plus court chemin de *depart* à u}
11: Mettre le sommet u dans l'ensemble *Atteint*
12: **for all** les sommets v voisins du sommet u **faire**
13: DistTemp := $pcc(u)$ + $cout(u, v)$;
14: **si** $v \notin$ *Atteint* ET DistTemp < $pcc(v)$ **alors**
15: {Si c'est plus court de passer par u pour aller de *depart* à v}
16: $Cout(v)$:= DistTemp ;
17: $pcc(v)$:= DistTemp ;
18: Ajouter le sommet v dans l'ensemble *Atraiter*
19: **fin si**
20: **fin for**
21: **fin si**
22: **fin tant que**
23: **si** le Cout du sommet *arrivee* != ∞ **alors**
24: retourner 1
25: **else**
26: retourner 0
27: **fin si**

Remarques : Dans cet algorithme, le coût $Cout(u)$ d'un sommet et la valeur du plus court chemin $pcc(u)$ d'un sommet sont identiques, mais c'est sur le $Cout(u)$ que les sommets sont choisis ligne 6.

3.2 Algorithme Astar ou A*

L'algorithme de Dijkstra construit de manière itérative tous les chemins optimaux issus du point de départ. À chaque étape, on inclut le sommet dont le coût est minimal, ce coût étant la valeur du plus court chemin depuis le départ. On sait que si le chemin optimal pour rejoindre le sommet d'arrivée doit passer par ce sommet courant, le coût final sera égal au coût de ce sommet PLUS le coût du chemin entre le sommet d'arrivée et lui. Avec l'algorithme de Dijkstra, on n'a aucune idée de la valeur de ce dernier coût. Au contraire, A* estime le coût du chemin restant entre le sommet courant et le sommet d'arrivée par une fonction heuristique. Il faut disposer d'une fonction plausible, ce qui est facile si les coûts sont des distances : la fonction heuristique peut être la distance euclidienne (ligne droite) ou la

fonction $(abs(x1-x2)+abs(y1-y2))/2$. Quel que soit le problème traité, cette fonction heuristique doit être admissible, c'est à dire qu'elle doit minorer la valeur réelle du coût du chemin, tout en l'approchant le plus possible.

L'algorithme est alors identique, excepté l'utilisation du coût estimé $Cout(v)$, si l'on doit passer par le sommet v et de la fonction **Heuristique** ligne 16.

Algorithm 2 Int A*(int depart, int arrivee, graph_t graphe)

Entrées depart, l'indice du sommet de depart
Entrées arrivee, l'indice du sommet d'arrivee
Entrées graphe, le graphe
1: Initialiser un ensemble de sommets *Atraiter*, vide
2: Initialiser tous les *pcc* et les *Couts* des sommets à ∞
3: Mettre le *pcc* et le *Cout* du sommet *depart* à 0
4: Ajouter le sommet *depart* a l'ensemble *Atraiter*
5: **tant que** l'ensemble *Atraiter* n'est pas vide **faire**
6: Rechercher et supprimer le sommet u de plus faible *cout* de l'ensemble *Atraiter*
7: **si** Si u==*arrivee* **alors**
8: on a trouve la solution, retourner 1 et fin de la fonction
9: **else**
10: {On a trouve le plus court chemin de *depart* à u}
11: Mettre le sommet u dans l'ensemble *Atteint*
12: **for all** les sommets v voisins du sommet u **faire**
13: DistTemp := $pcc(u)$ + Cout(u, v) ;
14: **si** $v \notin$ *Atteint* ET DistTemp $<$ $pcc(v)$ **alors**
15: {Si c'est plus court de passer par u pour aller de *depart* à v}
16: $Cout(v)$:= DistTemp + **Heuristique**(v, a) ;
17: $pcc(v)$:= DistTemp ;
18: Ajouter le sommet v dans l'ensemble *Atraiter*
19: **fin si**
20: **fin for**
21: **fin si**
22: **fin tant que**
23: **si** le cout du sommet *arrivee* $\neq \infty$ **alors**
24: retourner 1
25: **else**
26: retourner 0
27: **fin si**

Remarques :

— La fonction $Heuristique(v, a)$ estime le coût du chemin restant entre le sommet v et le sommet a. La fonction $(abs(x1-x2)+abs(y1-y2))/2$ convient pour les distances sur les réseaux routiers.

4 Choix des structures de données internes aux 2 algorithmes

— La notion d'*ensemble Atraiter* et *Atteint* est une notion générique : utilisez la structure de données la plus adaptée à la situation en termes de coût algorithmique : tableau, liste, liste triée, tas, arbre binaire de recherche, ou autre.

Les opérations à réaliser sur l'ensemble *Atraiter* sont l'ajout, la modification et la recherche du plus *petit*. Celles à réaliser sur l'ensemble *Atteint* sont l'ajout et la présence.

— Notez que l'ensemble *Atraiter*, quelle que soit la structure de donnée retenue, contient plutôt l'indice des sommets dans le tableau des sommets, ou un pointeur sur sommet, et non le sommet lui-même, pour ne pas dupliquer inutilement les données, d'une part et parce que son coût peut évoluer entre son entrée et sa sortie de l'ensemble, d'autre part.

— Optimisation : la ligne 18 ajoute le sommet v à l'ensemble *Atraiter*. Mais ce sommet peut déjà se trouver dans cet ensemble, avec un coût supérieur. Si la structure de données utilisée pour représenter cet ensemble est basée sur une relation d'ordre (liste triée, tas, arbre binaire de recherche), on peut optimiser l'espace mémoire et le temps en modifiant la position de ce sommet v dans notre ensemble en respectant sa nouvelle valeur de *cout*, plutôt que de simplement l'ajouter à la structure.

5 Reconstituer le chemin

Les 2 algorithmes précédents donnent la valeur du plus court chemin, mais pas le chemin lui-même. Pour le retrouver après l'exécution de l'algorithme, il faut stocker le prédécesseur de chaque sommet, au fur et à mesure que l'on modifie la valeur estimée de son plus court chemin depuis le sommet de départ. Cela se fait en insérant une instruction du type Pere(v) = u entre les lignes 17 et 18 des 2 méthodes. En effet, à cet endroit, on a trouvé un chemin qui mène au sommet v en passant par le sommet u, plus court que ceux qui pouvaient exister auparavant. Donc, le sommet u est peut-être le prédécesseur du sommet v sur le plus court chemin entre *depart* et *arrivee*.

Une fois trouvé le plus court chemin entre *depart* et *arrivee*, il suffit de remonter le chemin de Pere en Pere à partir du sommet *arrivee*. Le chemin est alors affiché à l'envers, de *arrivee* à *depart*. A vous de trouver une solution pour l'afficher dans le bon sens. [1]

6 Fichiers de données

Pour tester votre algorithme, nous fournissons plusieurs fichiers de données représentant des exemples très simples, le métro parisien et plusieurs parties du réseau routier américain.

6.1 Format des fichiers

Le format des fichiers est le suivant (visualisez-le en ouvrant l'un des fichiers exemple à l'aide d'un éditeur de texte comme `atom` ou avec la commande linux `cat graphe2.txt)` :

— Première ligne : deux entiers ; nombre de sommets (NBS) nombre d'arcs (NBA)

— Deuxième ligne : la chaîne de caractères "Sommets du graphe"

— NBS lignes dont le format est le suivant :

1. un entier : numéro du sommet [2]
2. deux réels indiquant la latitude et la longitude
3. une chaîne de caractères (sans séparateur) contenant le « nom de la ligne ». C'est toujours "M1", sauf dans le fichier du métro parisien : "M1", "M3bis", "T3", "A2", etc. par exemple.
4. une chaîne de caractères contenant le nom du sommet (qui peut contenir des séparateurs, par exemple des espaces).

— 1 ligne : la chaîne de caractères "arc du graphe : départ arrivée valeur"

— NBA lignes dont le format est le suivant :

1. un entier : numéro du sommet de départ
2. un entier : numéro du sommet d'arrivée
3. un réel : valeur ou coût de l'arc

Les fichiers contenant les graphes sont sur le site tdinfo.phelma.grenoble-inp.fr, sur les machines de l'école, dans le répertoire `/users/prog1a/C/librairie/projetS22021` et dans votre depot git pour les petits fichiers `graphe1.txt`, `graphe2.txt` et `metro.txt`.

— `graphe1.txt` et `graphe2.txt` contiennent les petits graphes de la figure 4.

1. La solution la plus simple est récursive et s'écrit en 5 lignes.
2. L'ordre des sommets est croissant dans le fichier.

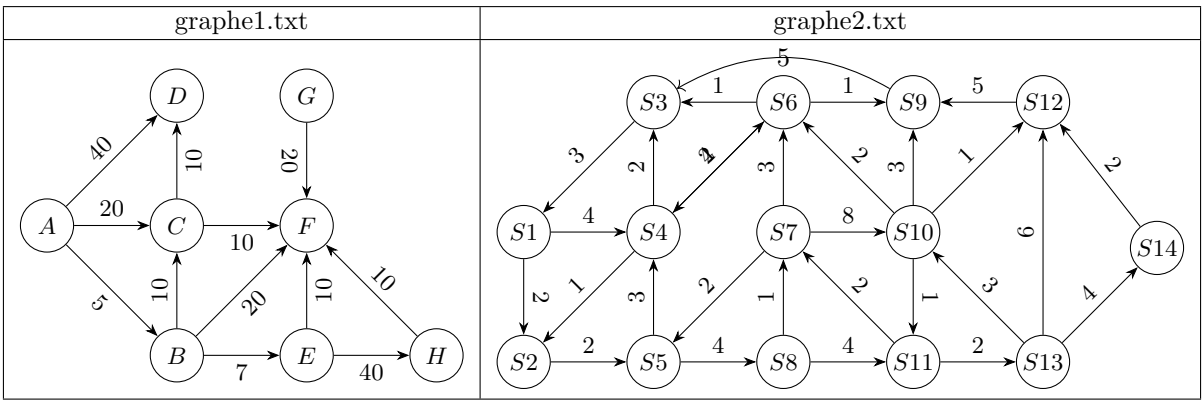

FIGURE 4 – Les 2 petits graphes.

— `metro.txt` contient le graphe représentant les lignes de métro et RER de Paris.

— Les fichiers `grapheAmericas.txt, grapheNewYork.txt, grapheColorado.txt, grapheMonde.txt, grapheFloride.txt, grapheGrandLacs.txt, grapheUSAOuest.txt et grapheUSACentral.txt` contiennent des extraits des réseaux routiers. Ne les recopiez pas dans vos répertoires !

Ces fichiers sont très volumineux et comportent de 250000 à 14000000 sommets. Ils permettent de tester la robustesse et l'efficacité de votre programme sur de gros volumes de données.

6.2 Lecture des fichiers en C

Pour rappel, on ne peut pas manipuler directement les données d'un fichier : on utilise un objet informatique intermédiaire de type `FILE *`, qui donne accès à ces données. On commence par ouvrir le fichier, c'est à dire associer l'objet de type `FILE *` à notre fichier réel, à l'aide de la fonction `fopen`. On peut ensuite lire les données du fichier pour les mettre dans des variables avec lesquelles notre programme peut travailler avec les fonctions `fscanf` ou `fread`, selon que ces données sont stockées sous forme textuelle ou binaire. Il faut enfin fermer le fichier avec la fonction `fclose`.

En pratique pour nos fichiers, la lecture des lignes contenant les sommets du graphe (les NBS lignes) doit se faire en lisant d'abord un entier, deux réels et une chaîne, puis une autre chaîne de caractères susceptible de contenir des espaces. Le plus simple est de lire les entiers et les réels ainsi que la ligne de métro avec la fonction `fscanf`, puis tout le reste de la ligne c'est à dire nom du sommet (chaîne avec séparateurs), avec la fonction `fgets`.

L'exemple ci dessous vous donne des éléments pour effectuer la lecture complète d'un fichier contenant un graphe. Il effectue l'ouverture du fichier à lire, lit le nombre de sommets et d'arcs, lit la ligne de commentaires "Sommets du graphe", lit les informations du premier sommet, saute une ligne et ferme le fichier.

```
FILE* f;
int numero,nbsommet,nbarc;
double lat,longi ;
char line[128] ;
char mot[512] ;

f=fopen("graphe1.txt","r");
if (f==NULL) { printf("Impossible d'ouvrir le fichier\n"); exit(EXIT_FAILURE);}
/* Lecture de la premiere ligne du fichier : nombre de sommets et nombre d'arcs */
fscanf(f,"%d %d ",&nbsommet,&nbarc);
/* Ligne de texte "Sommets du graphe" qui ne sert a rien */
fgets(mot,511,f);

/* lecture d'une ligne de description d'un sommet  */
/* on lit d'abord le numero du sommet, sa position et  le nom de la ligne */
fscanf(f,"%d %lf %lf %s", &numero, &lat, &longi, line);
/* numero contient alors l'entier ou numero du sommet, lat et longi la position, line le nom de la
   ligne */
```

```
/* Le nom de la station peut contenir des separateurs comme l'espace. On utilise plutot fgets pour lire
    toute la fin de ligne */
fgets(mot,511,f);
/* fgets a lu toute la ligne, y compris le retour a la ligne. On supprime le caractere '\n' qui peut se
    trouver a la fin de la chaine mot : */
if (mot[strlen(mot)-1]<32) mot[strlen(mot)-1]=0;
/* mot contient desormais le nom du sommet, espaces eventuels inclus. */

/*Pour sauter les lignes de commentaires, on peut simplement utiliser la fonction fgets, sans exploiter
    la chaine de caracteres lue dans le fichier */
fgets(mot,511,f);

/* Ne pas oublier de fermer votre fichier */
fclose(f);
```

Remarques :

— Le code précédent ne lit pas les arcs. Il reste à compléter et à adapter !

— Ce code ne vérifie pas non plus les cas d'erreur, par exemple le cas où le fichier ne débuterait pas par deux entiers, où il annoncerait 12 sommets mais n'en contiendrait que 11, où la chaine "Sommets du graphe" serait incorrecte, etc. Les plus valeureux d'entre vous s'attacheront à rendre leur fonction de lecture de fichier robuste, en protégeant ces cas d'erreur. Utiliser, par exemple, la valeur entière retournée par la fonction `scanf`, etc...

7 Autres ressources fournies

7.1 Fichiers de résultats

Pour faciliter la vérification de votre programme, nous fournissons les ressources suivantes dans le répertoire `/users/prog1a/C/librairie/projetS22021` des machines de l'école et sur le site des cours d'informatique `https://tdinfo.phelma.grenoble-inp.fr`.

Les fichiers suivants contiennent la valeur de tous les chemins existants dans les 2 petits graphes.

— `resultats_graphe1.txt` : contient la valeur de tous les chemins du `graphe1.txt`

— `resultats_graphe2.txt` : contient la valeur de tous les chemins du `graphe2.txt`

Ces fichiers ont été obtenus avec un programme testant tous les couples de sommets $(depart, arrivee)$ et une instruction d'affichage du type `printf("Plus court chemin entre le sommet %s et le sommet %s :%lf\n" ,graphe.data[depart].nom,graphe.data[arrivee].nom,graphe.data[arrivee].pcc);` Une fois réalisées les fonctions de lecture du graphe et de calcul de plus court chemin, il est facile de faire un programme réalisant la même chose. Si vous réalisez un tel programme, vous pouvez facilement comparer la sortie de votre programme avec les commandes `./prog > mesRes.txt; diff mesRest.txt resultats_graphe1.txt`

7.2 Programmes disponibles sur les machines de l'école

Une solution au plus court chemin peut aussi être obtenue en exécutant les commandes suivantes sur les machines de l'école.

— en mode texte par l'exécutable `pcc` qui donne la valeur du PCC et le temps d'exécution des 2 algorithmes avec la commande :

```
cd /users/prog1a/C/librairie/projetS22021; ./pcc grapheNewYork.txt
```

— en mode texte par l'exécutable `pcc` pour afficher le chemin avec la commande :

```
cd /users/prog1a/C/librairie/projetS22021; ./pcc grapheNewYork.txt -v
```

— en mode graphique par l'exécutable `pccgraph` pour visualiser le graphe et le résultat :

```
cd /users/prog1a/C/librairie/projetS22021; ./pccgraph grapheNewYork.txt 1
```

Les commandes précédentes demandent d'interagir avec le programme pour préciser les numéros des sommets de départ et d'arrivée. Pour aller plus loin et comparer plus de chemins, vous pouvez utiliser

un tirage aléatoire des sommets de départ et d'arrivée avec une instruction du type `depart=random()%graph.size_vertices`. Vous pouvez ainsi facilement calculer plusieurs chemins dont les départ et arrivée sont tirés aléatoirement et les comparer au résultat de la commande `pcc_rand`. Cette commande affiche la valeur des plus courts chemins et le temps de recherche d'un nombre de tirages précisés sur la ligne de commande (100 dans l'exemple ci dessous), ainsi que le temps de recherche moyen.

`cd /users/prog1a/C/librairie/projetS22021; ./pcc_rand grapheNewYork.txt 100 1` pour Astar

`cd /users/prog1a/C/librairie/projetS22021; ./pcc_rand grapheNewYork.txt 100 0` pour Dijkstra

7.3 Documentations diverses

Les documentations minimalistes pour utiliser les outils de développement sont toujours disponibles sur le site http://tdinfo.phelma.grenoble-inp.fr/1Apet/site/.

En particulier, vous trouverez les documentations sur le gestionnaire de version `git`, les déboggueurs `gdb` sur linux et lldb sur mac, l'utilitaire `valgrind`, les fichiers Makefile à l'adresse http://tdinfo.phelma.grenoble-inp.fr/1Apet/site/outils/outils.

8 Travail demandé

Le travail demandé consiste à programmer et à comparer les performances des 2 algorithmes. Le réseau routier permettra d'analyser les performances obtenues sur des grands volumes de données et de comparer aux performances théoriques, le métro d'introduire les notions de lignes et de correspondance entre lignes.

8.1 Réseau routier

L'objectif est de charger le graphe en mémoire à partir du fichier représentant le réseau routier, puis de demander à l'utilisateur un numéro de sommet de départ et un numéro de sommet d'arrivée, de calculer la valeur du plus court chemin et enfin d'afficher le chemin trouvé (i.e. : la série de sommets qui le constitue).

Il y a bien sûr des sens uniques : il n'existe pas toujours d'arc dans les 2 sens entre 2 sommets.

La particularité de cette étape est donc uniquement de travailler sur des graphes de taille importante : plus de 14 000 000 sommets et 34 000 000 arcs. Vos implémentations peuvent prendre plusieurs minutes ou heures selon les machines et la qualité de votre code. Assurez vous du bon fonctionnement de votre code sur des graphes simples avant de tester ceux-ci.

Assurez vous avec la commande valgrind que votre mémoire est bien gérée

8.1.1 Performances CPU et mémoire

Une fois le code de votre application correct, pour évaluer la qualité de votre code, vous pouvez mesurer la complexité pratique de votre logiciel en traçant le temps de calcul de la recherche du plus court chemin en fonction du nombre de sommets de vos graphes.

Bien évidemment, le temps dépend de la machine sur laquelle vous travaillez. L'implémentation disponible sur les machines de l'école vous donne une idée des temps de calculs d'une implémentation raisonnable.

Ce temps dépend aussi du chemin à trouver : un chemin entre 2 sommets voisins est plus facile à trouver qu'un chemin entre sommets plus éloignés. Evaluer le temps moyen doit se faire en recherchant un nombre de chemin raisonnables (quelques centaines) entre des sommets tirés au hasard grâce à la fonction `int random()`. L'instruction `depart=random()%graphe.size_vertices;` permet de tirer un numéro de sommet au hasard, en restant dans les limites du graphe.

Pour rappel, mesurer le temps pris par un appel de fonction peut se faire de la manière suivante :

```
// Ne pas oublier d'inclure time.h au debut pour la fonction clock()
#include <time.h>

int cl;
cl=clock();
// Appel de la fonction dont on veut mesurer le temps d'execution
f();
cl = clock()-cl;
printf("Temps mesure en secondes: %lf\n",cl/(double)CLOCKS_PER_SEC);
```

Les fichiers fournis contiennent :

Fichiers	Nb Sommets	Nb Arcs
USACentral	14081816	34292496
USAOuest	6262104	15248146
GrandsLacs	2758119	6885658
Floride	1070376	2712798
Monde	851205	2270007
Colorado	435666	1057066
NewYork	264346	733846
Americas	124049	371031

TABLE 1 – Nombres de sommets et d'arcs dans les fichiers.

Pour exemple, voici un tracé des performances en secondes sur les graphes routiers (figure 5). C'est le relevé du temps de calcul de chemins obtenus par les algorithmes de Dijkstra et A^* entre 500 sommets de départ et d'arrivée tirés au hasard.

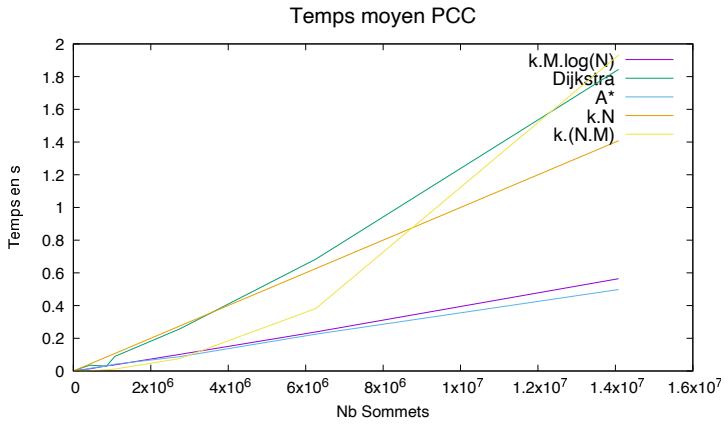

FIGURE 5 – Moyenne de 500 plus courts chemins.

Sur les fichiers proposés, comparez les résultats pratiques obtenus en moyenne sur quelques centaines de chemins tirés au hasard à la complexité théorique en fonction du nombre de sommets n. Commentez les résultats que vous obtenez.

Evaluer aussi la mémoire consommée par vos principales structures de données : le tableau des sommets, les listes d'arcs, les structures utilisées pour les ensembles *Atraiter* et *Atteint*.

8.2 Métro et RER parisien

Lorsque vous prenez le métro, il est rare de ne pas faire de correspondance pour changer de ligne. Ces correspondances sont prises en compte dans le graphe que nous fournissons.

Lorsque 2 (ou plus) lignes de métro se croisent avec correspondance entre lignes, il y a dans les fichiers un noeud pour chaque ligne, même si ces deux (ou plus) noeuds ont le même nom de station. Les noeuds du fichier métro parisien ne représentent pas une station (la Gare du Nord par exemple), mais un quai dans cette station (par exemple le quai du RERB de la Gare du Nord). Par exemple, on trouve un noeud Gare du Nord sur la ligne 4 (sommet numéro 84), la ligne 5 (sommet numéro 114), la ligne B (sommet numéro 457) et la ligne D (sommet numéro 580).

Cette notion de correspondance est prise en compte dans ce fichier : si deux lignes 1 et 2 se croisent avec correspondance, deux sommets différents existent pour cette station de correspondance, un sur la ligne 1 et un autre sur la ligne 2, et la correspondance est modélisée un arc de coût pré-établi, de valeur 360 entre ces deux sommets. Cet arc existe dans le fichier décrivant le métro.

Les correspondances à pied (exemple : entre Gare du Nord et La chapelle) ont un coût pré-établi de 600.

Le coût des autres arcs dépend du moyen de transport (métro, tramway et RER).

Dans un premier temps, vous désignerez les stations en utilisant leur numéro (un entier), c'est à dire que vous indiquerez sur quelle ligne vous démarrez votre voyage et vous afficherez explicitement les correspondances avec changement de ligne.

Lorsque cet affichage sera correct, vous réaliserez une (ou plusieurs) fonctions permettant à l'utilisateur de rentrer le nom (et non le numéro) de la station. Il faut alors faire correspondre le nom de la station (la chaîne de caractères) à tous les sommets de même nom, pour trouver tous les sommets dont le nom est "gare du nord" par exemple.

Pour trouver le meilleur chemin entre deux stations choisies par leur nom, il faut considérer le fait que chaque station peut être représentée par plusieurs noeuds dans le graphe. Par exemple, si vous partez de "Gare du Nord" pour aller à "Pantin", vous pouvez partir d'un des quatre sommets dont le nom est gare du nord (sommets 84,114,457 ou 580). Il faut donc :

- soit calculer le plus court chemin à partir de *chacun* de ces sommets de "Gare du Nord", donc appeler 4 fois votre fonction de recherche de plus court chemin avec chaque sommet de départ et le sommet Pantin (623) et choisir le meilleur des 4 chemins. Attention si le sommet d'arrivée comporte lui aussi plusieurs sommets avec son nom.
- soit définir un arc virtuel de coût nul entre chacun de ces sommets de départ. Ainsi, on peut passer du sommet 84 au sommet 114 avec un coût nul, comme si ces sommets étaient un seul et même sommet. On appelle une seule fois la fonction de recherche sur un des sommets, le premier trouvé par exemple. Il faudra supprimer les arcs virtuels de coût nul ensuite.
- soit concevoir toute autre solution à votre goût.

9 Réalisation

9.1 Première séance : analyse du problème

Cette séance permet de répondre à vos questions sur les algorithmes et les structures de données. Il faut bien sûr avoir lu et assimilé le sujet [3] puis constitué votre binôme.

A la fin de cette séance, vous devez avoir une vision claire des grandes étapes de votre programme et vous ferez un document décrivant :

- les types de données utilisées, en explicitant le rôle de chaque élément des structures
- les modules (couple de fichiers .c/.h) et le rôle de chaque module
- dans chaque module, les prototypes de fonctions **essentielles**, en explicitant :
  - le rôle exact de la fonction
  - le rôle de chaque paramètre et son mode de passage (par valeur ou par adresse)
  - l'algorithme ou les grandes étapes permettant de réaliser la fonction. Précisez uniquement les points qui peuvent être délicats à comprendre et/ou programmer, en quelques lignes au total.

3. ce que vous avez fait si vous lisez cela !

— les tests prévus
  — tests unitaires : tests des fonctions précédentes individuellement. Par exemple, il faut tester la fonction de lecture du graphe avant même d'essayer de calculer un chemin.
  — tests d'intégration : quels sont les tests que vous allez faire pour prouver que l'application fonctionne, sur quels exemples ?
— la répartition du travail entre les 2 membres du binôme : quelles sont les fonctions qui seront réalisées par chaque membre du binôme et pour quelle séance ?
— Le planning de réalisation du projet jusqu'à la date du rendu, le 30 mai 2021. Le dépôt gitlab sera fermé à ce moment. Prévoyez les jalons et les dates de réunion entre vous à l'avance.

9.2 Gestion du projet

Nous vous conseillons d'avoir un certain nombre de jalons et de prévoir leur réalisation à une date fixée à l'avance, que vous vous efforcerez de respecter. Voici un exemple de jalons que vous pourriez utiliser :

1. Mise en place des structures de données : prendre des structures identiques pour les 2 algorihtmes
2. Fonctions sur les listes
3. Lecture et affichage du graphe
4. PCC par Dijkstra, test petits fichiers
5. PCC par A*, test petits fichiers
6. Reconstitution du chemin (identique pour Dijkstra et A*),
7. Tests petits fichiers
8. Tests réseau routier
9. Tests métro sans affichage des correspondances
10. Affichage des correspondances du métro, changement de ligne
11. Gestion des stations par nom au lieu de numéro
12. Tests métro par nom de station
13. Optimisation des structures de données et du code

9.3 Conseils de développement

— Ne pas oublier le `git pull` quand vous commencez à travailler, et les `git add`,`git commit`, `git push` quand le code est validé
— Partagez vous les fichiers et évitez si possible de travailler sur les mêmes fichiers simultanément pour éviter les conflits sous `git`
— Essayer de prévoir un ordre logique dans la réalisation de vos fonctions : par exemple, il faut commencer par écrire la fonction de lecture du graphe et la vérifier. Les sommets contiennent les voisins qui sont des listes d'arcs. Il faut donc commencer par écrire et tester les fonctions sur les listes d'arcs (création, ajout, visualisation). Ensuite, il faut écrire la fonction de chargement du graphe, celle d'affichage du graphe et un programme de test de ces 2 fonctions.
— Partagez vous le travail en fonction de vos compétences afin de ne pas ralentir le déroulement du projet.
— Mettez régulièrement le travail en commun.
— Compilez souvent, par exemple chaque fois qu'une fonction est écrite : évitez de vous retrouver avec beaucoup de code plein d'erreurs de compilation.
— Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure. Ne pas écrire tout le code et compiler puis tester au dernier moment.
— Validez le programme réalisé sur les graphes simples `graphe1.txt` et `graphe2.txt` avant de le tester sur des graphes plus conséquents.
— Utilisez `valgrind` pour vérifier que la mémoire est bien gérée.

9.4 Rendu final

Le rendu final se fait sur votre dépôt git commun aux deux membres du projet, **sans les exécutables ni les fichiers de données**, le 30 mai 2021 au plus tard. Le livrable sera constitué :

— du rapport du projet, format PDF. N'oubliez pas de faire figurer vos noms, date, contexte sur la page de garde...

— des fichiers sources de votre programme

— du fichier Makefile

— d'un fichier README expliquant comment compiler et lancer votre (vos) programme(s)

— Les binaires (*.o) et les exécutables **ne doivent pas** se trouver sur votre dépôt.

Vous ferez un rapport court (5 à 10 pages) explicitant les points suivants :

1. Implantation
   (a) État du logiciel : ce qui fonctionne, ce qui ne fonctionne pas
   (b) Justifier les structures de données que vous avez choisi (excepté celles données pour les graphes)
   (c) Tests effectués et exemple d'exécution
   (d) Analyse des performances CPU et mémoire, théorique et pratique
2. Suivi
   (a) Organisation au sein de l'équipe, Réunions et Planning effectif, qui a fait quoi.
   (b) Outils de développement utilisés
   (c) Problèmes rencontrés et solutions
   (d) Qu'avons nous appris et que faudrait il de plus ?
3. Conclusion

Notation

La notation de ce projet tiendra compte des résultats obtenus, du rapport (y compris la véracité des informations), de la qualité du code, des procedures de test, des outils de développement et de leur maitrise, du respect du cahier des charges, de l'assiduité en séance.

Tout plagiat, qu'il soit interne ou externe à Phelma, est bien sûr interdit et sera sanctionné.

10 Extensions

10.1 Recherche des stations par leur nom

La recherche des stations par leur nom, au lieu de leur numéro, peut être facilitée par une table de hachage, avec gestion des collisions par listes chaînées. En effet, tous les sommets qui ont le même nom seront alors dans la même liste chaînée, puisque le hachage se fait sur le même nom. Cette liste est très réduite par rapport au nombre total de sommets et permet une recherche rapide. Notez que, comme d'habitude avec les tables de hachage, d'autres noms de station peuvent cependant se retrouver dans cette liste de collisions, si la fonction de hachage de 2 noms de station différents donne le même indice entier.

10.2 Version graphique

En exploitant les coordonnées GPS des différents sommets, vous pouvez dessiner le graphe et proposer une interface Homme Machine autre que clavier et écran texte. Seule la bibliothèque SDL est autorisée, afin que nous puissions compiler et vérifier votre code.