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.

L'objectif du projet est de calculer le meilleur itinéraire, i.e. le plus court chemin entre 2 villes ou 2 stations de métro parisien. Le réseau routier, ou le métro, se représente facilement par un graphe, chaque sommet étant une intersection de routes ou une station de métro, les arcs représentant les routes et la distance, ou une ligne de métro et la distance entre 2 stations.

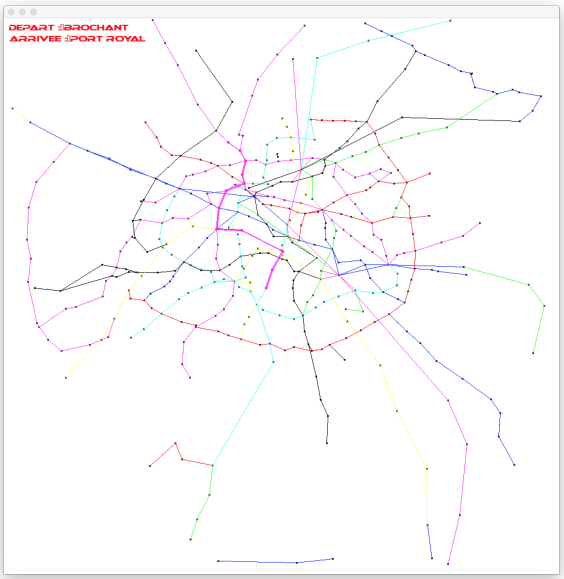


FIGURE 1 – Réseau et plus court chemin entre Brochant et PortRoyal

2 Graphes

2.1 Terminologie et définitions

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

Arc (*edge* en anglais)= arête orientée.

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

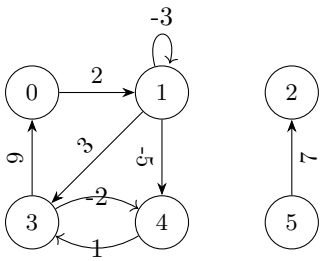


FIGURE 2 – Un exemple de graphe

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 (*neighbour*) : les voisins d'un sommet sont ceux qui lui sont reliés par un arc.

2.2 Représentation des sommets et des arcs

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 et la liste de ses successeurs. Un arc est une structure contenant le numéro du sommet d'arrivée et le coût de l'arc.

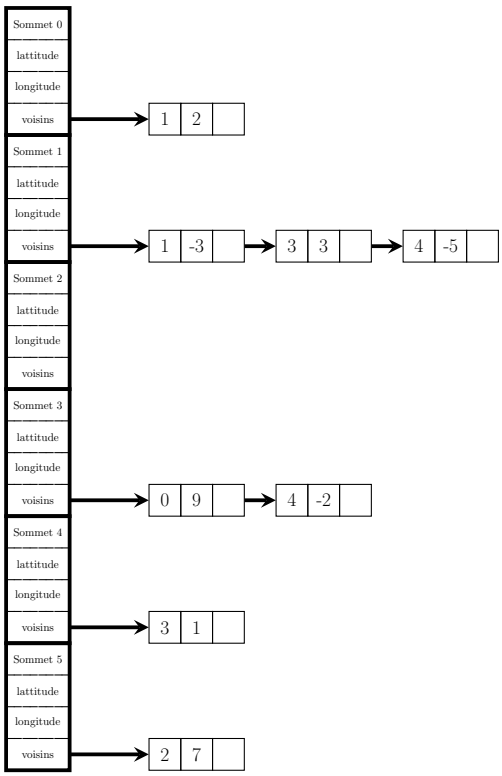


FIGURE 3 – Représentation du graphe de la figure 2 par liste chainée de successeurs.

La liste des successeurs d'un sommet est une liste chaînée des arcs qui partent de ce sommet. On utilisera une liste chaînée dynamique, comme vu en cours. Les types de données ressembleront à :

```
// Type arc
typedef struct {
   int depart, arrivee;   // Les indices des sommets de depart et d'arrivee de l'arc
   double cout;           // Le cout 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 donne au sommet
    char* ligne;        // Nom de la ligne, uniquement pour pour le metro
    double x,y;         // coordonnees latitude et longitude du sommet pour le graphqie
    listedge_t edges;   // Liste des arcs qui partent de ce sommet
    double pcc;         // "plus court chemin" entre le sommet de depart et ce sommet.
                        // N'est utile que durant le déroulé de l'algorithme.
} vertex_t;

// Type graphe :
typedef struct {
  int size_vertex;    // Nombre de sommets
  int size_egdes;     // Nombre d'arcs
  vertex_t* data;     // Tableau des sommets alloue dynamiquement
} graph_t;
```

2.3 Représentation du graphe

Un graphe se représente alors comme un tableau de sommets (voir figure 3) et le nombre de ces sommets. Chaque sommet contient les informations définies ci dessus, auxquelles on pourra si nécessaire ajouter d'autres éléments utiles aux algorithmes décrits dans les parties suivantes.

3 Plus court chemin dans un graphe

Dans un problème de plus court chemin (PCC), on considère un graphe orienté $\mathcal{G} = (X, A)$. Chaque arc a_i 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 bien sur 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 bien sur du problème traité.

— L'algorithme de *Dijkstra*, dans le cas d'arcs à valuation positive, est celui utilisé dans le routage des réseaux IP/OSPF.

— 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 *A** ou *Astar* utilise une heuristique pour trouver rapidement une solution approchée. Il ne s'applique qu'aux graphes dont les arcs sont à valuation positive. Il nécessite de disposer d'une fonction heuristique pour estimer la distance entre deux nœuds.

Tous les algorithmes s'appuient sur la notion de *coût d'un sommet* : durant l'algorithme, pour chaque sommet, on stocke le coût du meilleur chemin découvert pour atteindre ce sommet depuis le sommet de départ. Au début, les coûts de tous les sommets sont initialisés à l'infini, sauf celui de départ qui est initialisé à 0. Lorsqu'un premier chemin vers un sommet est trouvé, on va stocker le coût de ce chemin dans le sommet. Puis, au fur et à mesure de l'algorithme, à chaque fois qu'un meilleur chemin vers ce sommet est découvert, on met à jour le coût du sommet.

3.1 Algorithme Depth Fisrt Search ou DFS

Dans un graphe, plusieurs chemins possibles relient 2 sommets. Trouver *un* chemin entre 2 sommets n'est donc pas suffisant. DFS doit examiner *tous* les chemins pour déterminer le meilleur.

L'algorithme est décrit page suivante.

Il s'écrit facilement de manière récursive. A partir du sommet de départ, on examine le premier voisin. Pour atteindre ce voisin en passant par le sommet de départ, le coût du chemin est donc le coût du sommet de départ auquel au ajoute le coût de l'arc entre le sommet de départ et le voisin examiné. Si le voisin a déjà été atteint par un autre chemin, il a déjà un coût. On compare ce coût au nouveau coût que l'on vient de calculer et on choisit le meilleur des deux.

C'est le processus récursif qui va être utilisé pour trouver le plus court chemin.

Lorsque l'on examine un noeud, on choisit le premier voisin. Si le coût du chemin actuel de ce voisin est supérieur au coût du sommet examiné + coût de l'arc entre le sommet examiné et ce voisin, on a trouvé un meilleur chemin pour atteindre ce voisin. On met à jour son coût et on appelle la fonction récursivement sur ce sommet. On choisit ensuite le deuxième voisin, puis le troisième, etc...

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

Entrées: depart : indice du sommet de depart. Le sommet de depart doit exister.
Entrées: arrivee : indice du sommet d'arrivee. Le sommet d'arrivee doit exister.
Entrées: graphe : le graphe. Tous les coûts des arcs doivent être positifs.

```
1:  Si depart == arrivee Alors
2:    retourner 1
3:  Sinon
4:    trouve :=0
5:    Pour tout  arc (depart, voisin) du graphe Faire
6:      DistTemp := Cout(depart) + Cout de l'arc(depart, voisin) ;
7:      Si  DistTemp < Cout(voisin)  Alors
8:        Cout(voisin) := DistTemp ;
9:        trouve=ParcoursEnProfondeur(voisin, arrivee, graphe) ;
10:     Fin Si
11:   Fin Pour
12:   retourner trouve
13: Fin Si
```

Remarques :

1. Avant d'appeler cette fonction, il faut initialiser les valeurs des coûts des sommets à l'infini et celui du départ à ZERO.
2. `Cout(sommet)` correspond en fait simplement le champ `pcc` de la structure `vertex_t`.
3. L'algorithme présenté donne la valeur du plus court chemin. Pour retrouver le chemin lui-même, il faut ajouter une information aux sommets indiquant par quel *sommet pere* le chemin arrive à ce sommet. Cette information sur le prédécesseur d'un sommet dans le chemin le plus court est mis à jour en même temps que la mise à jour de la distance à la ligne 8, avant l'appel récursif.
4. Lorsque l'algorithme est terminé, pour reconstruire le meilleur chemin, on part du sommet d'arrivée, et on remonte de prédécesseur en prédécesseur les pères, jusqu'au sommet de départ. La série de sommets ainsi constituée est le meilleur chemin.
5. On peut améliorer cet algorithme en arrêtant la récursivité pour les arcs (depart, voisin) dont la distance est déjà supérieure à la distance inscrite dans le sommet d'arrivée

3.2 Algorithme Breadth Fisrt Search ou BFS

L'algorithme BFS consiste à partir du sommet de départ, puis à examiner tous les voisins de ce sommet, puis tous les voisins des voisins, etc... Il est assez efficace pour trouver UN chemin. L'algorithme s'écrit à l'aide d'une file **d'indices** de sommet, ou de **pointeurs** sur sommet, pour gérer correctement l'ordre d'examen des sommets. Il ressemble en cela au parcours au largeur d'un arbre, vu en cours.

L'algorithme BFS est décrit page suivante.

Remarques :

— Pour éviter d'ajouter plusieurs fois un sommet dans la file (ligne 11, page suivante), on peut ajouter une information dans la structure `vertex_t` indiquant si le sommet a déjà été ajouté ou non.

— Lorsque l'on défile un sommet, ligne 6, il est inutile de traiter les voisins de ce sommet si le coût de ce sommet est déjà supérieur au meilleur chemin actuel du sommet arrivée. On élimine ainsi beaucoup de chemins qui seront de toutes façons trop long.

— Notez bien que chaque maillon de la file contient l'indice d'un sommet dans le tableau des sommets, ou un pointeur sur sommet, et non le sommet lui-même, car son coût peut évoluer entre son entrée et sa sortie de la file !

Algorithm 2 Int ParcoursEnLargeur(int depart, int arrivee, graph_t graphe)

Entrées: depart, indice du sommet de depart
Entrées: arrivee, indice du sommet d'arrivee
Entrées: graphe, le graphe

```
 1: Initialiser une file vide f de sommets
 2: Initialiser tous les couts des sommets à ∞
 3: Mettre le cout du sommet de depart à 0
 4: Enfiler le sommet de depart sur la file f
 5: Tant que la file f n'est pas vide Faire
 6:    Defiler le sommet u de la file f
 7:    Pour tout  arc (u,v) du graphe Faire
 8:       DistTemp := Cout(u) + Cout(u,v) ;
 9:       Si  DistTemp < Cout(v)  Alors
10:          Cout(v) := DistTemp ;
11:          Si v n'est pas dans la file Alors
12:             enfiler v dans la file f
13:          Fin Si
14:       Fin Si
15:    Fin Pour
16: Fin Tant que
17: Si cout du sommet arrivee != ∞ Alors
18:    retourner 1
19: Sinon
20:    retourner 0
21: Fin Si
```

3.3 BFS amélioré

L'algorithme précédent examine en largeur tous les chemins partant du sommet de départ. L'ordre d'examen des sommets est l'ordre des voisins : voisins du premier ordre, puis du deuxième ordre, puis du troisième ordre, etc... : c'est le rôle de la file.

Ce choix n'est pas forcément optimal. Remplacer cette file par un tas *minimier* (voir le TAD heap) ordonné selon les coûts des sommets est bien plus efficace.

L'algorithme BFS ainsi amélioré est décrit page suivante.

En remplaçant cette file par un tas, le choix du prochain sommet se fait ligne 6 en prenant le sommet le moins couteux parmi tous ceux qui sont atteignables actuellement. Intuitivement, dans un graphe à valuations positives, on construit ainsi toujours le chemin de coût minimal entre le sommet de départ et les sommets qui *sortent* du tas. Dès que le sommet d'arrivée est extrait du tas, on est sûr d'avoir trouvé le plus court chemin.

En remplaçant la file par un tas, l'algorithme est proche de celui de Dijkstra. Il s'écrit de manière identique au BFS, excepté le test d'arrêt ligne 5, qui doit tester si le sommet d'arrivée est atteint. On limite fortement ainsi la complexité moyenne.

Remarques :

— La ligne 14 précisant de modifier la position de v dans le tas consiste à remonter le sommet v au sein du tas, pour garantir la propriété de tas.

En effet, si un meilleur chemin a été trouvé ligne 9, le coût du sommet est modifié. Le nouveau coût a toujours une valeur plus petite que celle que ce sommet avait auparavant. Dans le tas, il est donc possible que son père ne soit plus inférieur à ce fils. Dans ce cas, il faut remonter le sommet dans le tas pour reconstruire la propriété de tas. C'est le même mécanisme que lors de ajout d'un sommet au tas qu'il faut appliquer : on compare au père et on échange si ce n'est pas dans le bon ordre, et recommence avec le père tant que cet ordre n'est pas respecté.

— Pour retrouver rapidement à quel endroit un sommet donné est placé dans le tas, on peut ajouter un champ à la structure sommet qui indiquera l'indice du sommet dans le tas.

Algorithm 3 Int Dijkstra_like(int depart, int arrivee, graph_t graphe)

Entrées: depart, indice du sommet de depart
Entrées: arrivee, indice du sommet d'arrivee
Entrées: graphe, le graphe

```
1: Initialiser un tas vide t de sommets
2: Initialiser tous les couts des sommets à ∞
3: Mettre le cout du sommet de depart à 0
4: Ajouter le sommet de depart au tas t
5: Tant que le tas t n'est pas vide ET le sommet d'arrivee n'est pas atteint Faire
6:    Extraire le sommet u du tas t //Si u==arrivee, on a trouve la solution
7:    Pour tout  arc (u,v) du graphe Faire
8:       DistTemp := Cout(u) + Cout(u,v) ;
9:       Si  DistTemp < Cout(v)  Alors
10:         Cout(v) := DistTemp ;
11:         Si v n'est pas dans le tas Alors
12:            Ajouter v au tas t
13:         Sinon
14:            Modifier la position de v dans le tas t
15:         Fin Si
16:      Fin Si
17:   Fin Pour
18: Fin Tant que
19: Si cout du sommet arrivee != ∞ Alors
20:    retourner 1
21: Sinon
22:    retourner 0
23: Fin Si
```

— Au lieu de modifier le tas ligne 14, on peut simplement ajouter à nouveau ce sommet dans le tas. Comme il a une valeur inférieure à la précédente, il sera extrait avant l'autre version du sommet. Il faut alors prévoir un tas qui peut contenir beaucoup plus de sommets que le nombre de sommets du graphe. L'ordre de grandeur est le nombre de sommets du graphe multiplié par le degré (nombre moyen de voisins d'un sommet). C'est une solution plus facile à implanter, mais plus couteuse en mémoire et moins élégante.

4 Fichiers de données

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

4.1 Format des fichiers

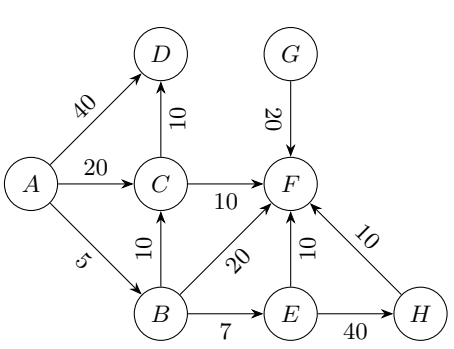
Le format des fichiers est le suivant (visualisez-le en ouvrant l'un des fichiers exemple) :

— 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 :

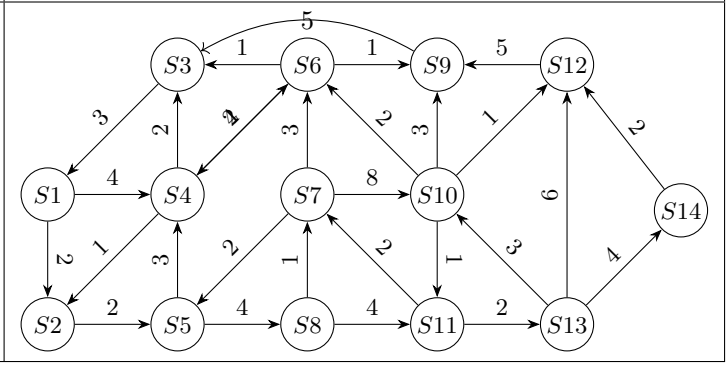
1. un entier : numéro du sommet (remarque : ils sont toujours donné dans l'ordre dans le fichier)
2. deux réels indiquant la latitude et la longitude
3. une chaîne de caractères (sans séparateur) contenant le « nom de la ligne ». C'est toujours "M1", sauf dans le fichier du métro parisien : "M1", "M3bis", "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

graphe1.txt



graphe2.txt



Les fichiers contenant les graphes sont sur les sites chamilo et tdinfo.phelma.grenoble-inp.fr et sur, sur les machines de l'école, dans le répertoire `/users/prog1a/C/librairie/projetS22020`.

— `graphe1.txt` et `graphe2.txt` contiennent les petits graphes ci-dessus
— `metro.csv` contient le graphe représentant les lignes de métro et RER de Paris
— Les fichiers `grapheNewYork.csv, grapheColorado.csv, grapheFloride.csv, grapheGrandLacs.csv, grapheUSAOuest.csv, grapheUSACentral.csv` contiennent des extraits du réseau routier américain.
  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.

4.2 Lecture des fichiers en C

Pour rappel, on ne peut pas manipuler directement les données d'un fichier : on utilise un objet informatique intermédiaire `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 textuelles 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 qui peut 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 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 sommet 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 numero du sommet, la position,  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. */

/*Pour sauter les lignes de commentaires, on peut simplement utiliser la fonction fgets sans utiliser
   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. A compléter et adapter dans une fonction !

— Ce code ne vérifie pas bien non plus les cas d'erreur, par exemple le cas où le fichier ne débuterait pas par deux entier, où il annoncerait 12 sommet 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.

5 Travail demandé

Le travail demandé consiste à programmer l'algorithme de recherche en profondeur d'abord ET l'algorithme de recherche en largeur ou l'algorithme de recherche en largeur amélioré.

Il se déroulera en 2 étapes :

1. une première étape consiste à déterminer le plus court chemin sur le réseau routier
2. une deuxième étape consiste à déterminer le plus court chemin sur le métro, qui comporte des correspondances

Bien sûr, commencer par vérifier que votre algorithme fonctionne correctement sur les petits graphes exemple - vous pouvez écrire d'autres fichiers exemples.

Vous pouvez obtenir une solution en vous connectant (avec vpn) à la machine `markov.phelma.grenoble-inp.fr` par la commande `ssh -X monlogin@markov.phelma.grenoble-inp.fr`. Voir la document d'accès au VPN sur le site `https://tdinfo.phelma.grenoble-inp.fr`, et la documentation de votre machine (Mac, Linux, Windows...) pour la commande ssh.

Une fois connecté en ssh, vous êtes sur une machine de Phelma, dans votre espace de fichiers habituel, dans une fenetre Terminal. Vous pouvez alors utiliser toutes les commandes `Unix` habituelles : `cd, mkdir, ls, clang, make`, etc...

Une solution au plus court chemin peut être obtenue :

— en mode texte par l'exécutable `pcc` avec la commande :
`cd /users/prog1a/C/librairie/projetS22020; ./pcc grapheNewYork.csv`

— en mode graphique par l'exécutable `pccgraph` pour visualiser le graphe et le résultat :
`cd /users/prog1a/C/librairie/projetS22020; ./pccgraph grapheNewYork.csv 1`

Tous les fichiers de test sont dans le répertoire `/users/prog1a/C/librairie/projetS22020` sur les machines de l'école, ainsi que sur le site `tdinfo.phelma.grenoble-inp.fr` et le site Chamilo du module.

Si vous êtes travaillez sur les machines de Phelma en ssh, **ne copiez pas les fichiers de test dans votre répertoire** car ils sont très gros.

Si vous travaillez sur votre machine personnelle, vous devrez bien sûr les télécharger.

5.1 Réseau routier américain

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 ce plus 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 très 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.

5.2 Metro et RER Parisien

Le graphe est ici un peu particulier : lorsque 2 (ou plus) lignes de métro se croisent (correspondance entre lignes), il y a dans les fichier un noeud pour chaque ligne, même si ces deux (ou plus) noeuds ont le même nom de station. Ainsi, d'une certaine manière, 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 metro. 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). Ensuite, 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 cout nul, comme si ces sommets étaient un seul sommet. On appelle une seule fois la fonction de recherche sur un des sommets. Il faudra supprimer les arcs virtuels de cout nul ensuite.
- soit concevoir toute autre solution à votre goût.

6 Réalisation

6.1 Première séance : analyse du problème

Cette séance permet de répondre à vos questions sur les algorithmes et les structures de données.

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.
- 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.

6.2 Conseils de développement

- 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 : éviter de se 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.

6.3 Rendu final

Chaque membre du binome copiera l'ensemble des fichiers constituant le livrable dans un fichier compressé sur chamilo, **sans les exécutables ni les fichiers de données**. 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 sources de votre programme
- du fichier Makefile
- d'un fichier README expliquant comment compiler et lancer votre (vos) programme(s)

— Vérifiez que vous n'avez PAS mis les fichiers de données, ni de fichiers binaires ou exécutables dans l'archive avant de la déposer

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) Tests effectués et Exemple d'exécution
2. Suivi
   (a) Problèmes rencontrés
   (b) Planning effectif, qui a fait quoi.
   (c) Qu'avons nous appris et que faudrait il de plus ?
3. Conclusion

7 Extensions

7.1 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 chainées. En effet, tous les sommets qui ont le même nom seront alors dans la même liste chainée, 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.

7.2 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.