# Projet : Plus Court Chemin dans un graphe et Algorithme $A^*$

**Notions :** Graphes

## 1 Introduction

Ce projet sera réalisé en binôme, sur les 4 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.

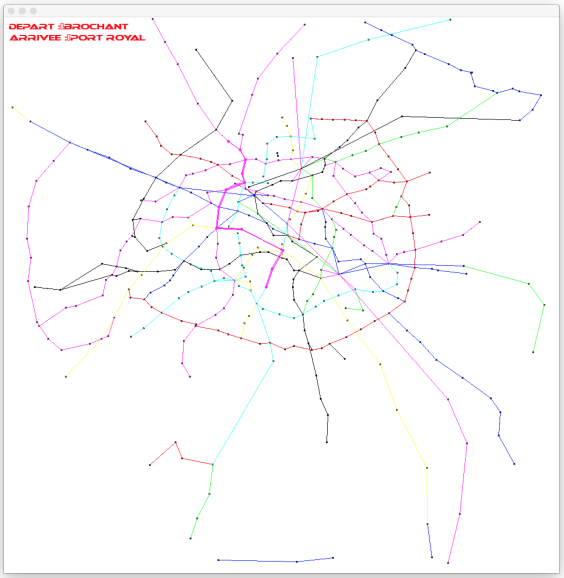



FIGURE 1 – Réseau et de plus court chemin

## 2 Graphes

### 2.1 Terminologie et définitions

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

**Arc** = arête orientée.

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

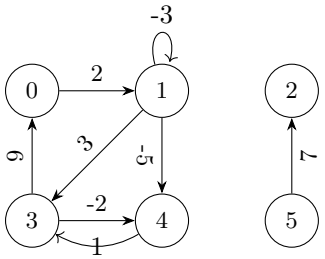



FIGURE 2 – Un exemple de graphe

**Circuit** = chemin dont les sommets de départ et d'arrivée sont identiques.

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

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

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

### 2.2 Représentation des sommets et des arcs

On peut représenter un sommet par une structure contenant son nom, sa latitude, sa longitude (pour dessiner le graphe), la ligne de métro à laquelle ce sommet appartient si besoin et la liste de ses successeurs possibles, c'est à dire la liste des arcs menant aux intersections de routes pour le réseau routier ou aux stations suivantes pour le métro. Un arc est représenté par une structure contenant le numéro du sommet

d'arrivée et le coût de l'arc. La liste des successeurs d'un sommet est alors la liste des arcs qui partent de ce sommet, une cellule contenant un arc comme valeur et un pointeur vers le suivant. Les types de données utiles ressembleront à :

**Les sommets :** T_SOMMET est une structure

```
typedef struct { char* nom; double x,y ; L_ARC voisins;} T_SOMMET ;
```

**Les arcs :** un arc est une structure

```
typedef struct { int arrivee; double cout; } T_ARC ;
```

**Les listes de successeurs :** une liste classique avec un suivant, la valeur est un arc.

```
typedef struct lsucc { T_ARC val; struct lsucc* suiv ; }* L_ARC;
```

### 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 peut ajouter d'autres éléments utiles aux algorithmes décrits dans les parties suivantes.

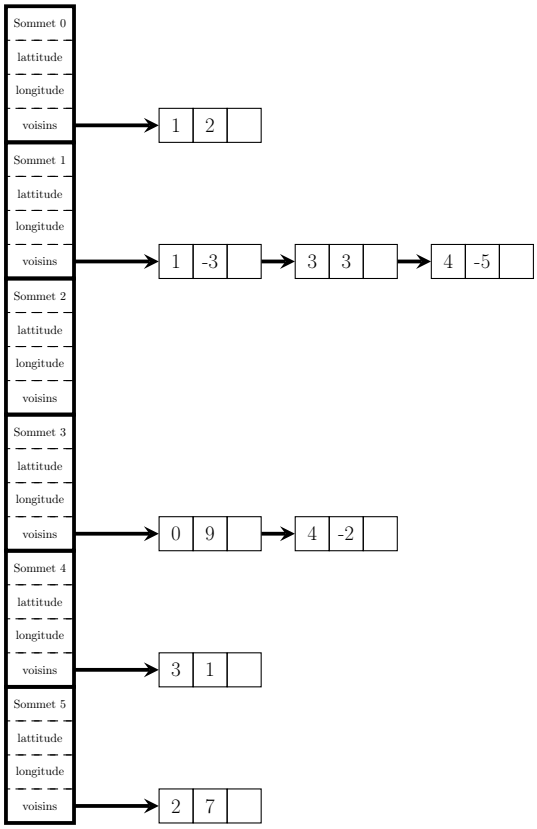



FIGURE 3 – Représentation du graphe précédent par liste chainée de successeurs.

## 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 $a_i$ 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 largeur d'abord permet de trouver le plus court chemin dans un graphe.

— 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 s'appliquer dans le cas d'arc 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 qui 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.

## 3.1 Algorithme A*

*Remarque préliminaire : cet algorithme est souvent utilisé pour la recherche de chemin dans un labyrinthe, sous le nom de pathfinding. Dans ce cas, la notion de graphe est implicite, i.e. il n'y a pas de graphe avec des listes de successeurs. Les codes que vous trouverez sur le www ne sont donc pas du tout adaptés à ce problème et il est inutile de les recopier.*

Le problème est de trouver le plus court chemin entre un sommet départ $d$ et un sommet arrivée $a$. Pour cela, on va "tracer" progressivement un chemin partant de $d$, en ajoutant le sommet qui, à chaque étape, s'approche au mieux de $a$. Lorsque que l'on se trouve sur un sommet $s$, l'algorithme $A*$ choisit le sommet $k$ (parmi les voisins de $s$) qui nous « approche le plus » du sommet $a$. Pour cela, si on traite le sommet $s$, on sépare le coût $F(s)$ du chemin entre $a$ et $d$ et passant par $s$ en 2 parties : $F(s) = G(s) + H(s, a)$.

1. $G(s) = cout(d, s)$ : représente le coût du chemin le plus court entre $d$ et $s$. Il est calculé pour tous les sommets au fur et à mesure de l'algorithme.
2. $H(s, a)$ estime le coût du chemin entre $s$ et $a$. Le coût réel du plus court chemin entre $s$ et $a$ étant inconnu, on utilise une heuristique qui minore le chemin réel. Par exemple, si les sommets $s$ et $a$ sont de coordonnées $x1, y1$ et $x2, y2$, la fonction $H$ peut être la distance euclidienne (ligne droite) ou la fonction $(abs(x1 - x2) + abs(y1 - y2))/2$, qui minore la distance euclidienne. L'heuristique doit être admissible, c'est-à-dire que $H(s, a) \leq cout(s, a)$, le véritable coût pour aller de $s$ vers $a$ : elle ne surestime jamais la vraie distance, auquel cas $A*$ trouvera le chemin optimal. Si de plus elle est consistante, c'est-à-dire que pour tout $y$ successeur de $s$, $H(s, a) \leq H(y, a) + cout(s, y)$, la complexité temporelle est linéaire.

L'algorithme revient à effectuer un parcours du graphe ordonné selon les valeurs de $F(s)$. On explore les sommets du graphe en prenant les sommets les uns après les autres, en choisissant toujours le sommet de valeur $F(s)$ minimale.

Tout au long de l'algorithme, on va utiliser et maintenir deux listes :

1. La liste fermée $LF$, qui contient les sommets $s$ du graphe qui ont été atteints, traités et dont le plus court chemin depuis $d$ est connu et vaut $G(s)$
2. La liste ouverte $LO$, qui contient les sommets du graphe dont un voisin a déjà été atteint (on connaît la valeur du plus court chemin pour ce voisin). Ces sommets sont donc accessibles depuis $d$ en passant par un voisin, mais il n'est pas sûr que cela soit le meilleur chemin. Cette liste sera ordonnée en fonction de la valeur $F(s)$ des sommets qui la composent.

Par ailleurs, il faut aussi conserver en mémoire le « meilleur chemin » trouvé entre le départ $d$ et chacun des sommets que l'on a déjà considérés. Pour cela, chaque sommet $s$ de ces listes doit simplement connaître son prédécesseur $Pere(s)$ par le meilleur chemin trouvé. Pour reconstruire un chemin, il suffit alors de partir de la fin, le sommet $a$ et de remonter les pères jusqu'au sommet de départ $d$.

L'algorithme A* pour aller du sommet $d$ au sommet $a$ peut se décrire de la manière suivante :

**Initialisation**
$G(d) \leftarrow 0$
Calculer $H(d, a)$
$F(d) \leftarrow G(d) + H(d, a)$
**for** tous les autres sommets k **do**
    $F(k) \leftarrow G(k) \leftarrow H(k, a) \leftarrow \infty$

**end for**
$LO \leftarrow \{d\}; LF \leftarrow \emptyset$ ▷ La liste ouverte est initialisée avec le sommet $d$ et la liste fermée est vide
$k \leftarrow d$
**Algorithme itératif**
**while** $LO \neq \emptyset$ & $k \neq a$ **do** ▷ la liste ouverte n'est pas vide ET l'arrivée n'est pas atteinte
$k \leftarrow i$ tel que $i \in LO$ & $\forall j \in LO\ F(i) \leq F(j)$ ▷ Extraire de la liste ouverte le sommet $k$ de plus faible valeur $F(k)$
**if** k==a **then** ▷ on a atteint l'arrivée et on a trouvé le plus court chemin. FIN.
**else**
// k est considéré comme le sommet atteignable le plus intéressant
// car nous rapprochant le plus de $a$ (au sens de l'heuristique choisie).
// On va donc avancer d'un sommet, en considérant qu'il faut passer par $k$
$LF = LF \cup \{k\}$ ▷ On met le sommet $k$ dans la liste fermée
// $k$ est atteint, la valeur $G(k)$ est le plus court chemin entre $d$ et $k$
// maintenant, il faut mettre à jour la liste ouverte, en considérant les nouveaux
// sommets atteignables à partir du sommet $k$
**for** $s \in sommets[k].voisins$ & $s \notin LF$ **do** ▷ sommets voisins de $k$ absents de la liste fermée
**if** $s \notin LO$ **then** ▷ $s$ n'a jamais été vu
// C'est un chemin possible ; Le père de $s$ est $k$
// Ajouter ce sommet à la liste ouverte.
// Il est possible qu'il soit le plus court pour aller de $d$ à $s$
$Pere(s) \leftarrow k$
$G(s) \leftarrow G(k) + cout(k, s)$
Calculer $H(s, a)$
$F(s) \leftarrow G(s) + H(s, a)$
$LO \leftarrow LO \cup \{s\}$ ▷ Insérer le sommet dans la liste ouverte au bon endroit
**else**
// Si $s$ est déjà dans la liste ouverte, un chemin passant par $s$ existe et $G(s)$ existe
// On regarde si le nouveau chemin est meilleur que l'ancien
**if** G(k)+cout(k,s) < G(s) **then** ▷ meilleur chemin de $d$ à $s$
// On vient donc de trouver un meilleur chemin de $d$ à $s$,
// plus court que celui obtenu précédemment
$Pere(s) \leftarrow k$
$G(s) \leftarrow G(k) + cout(k, s)$
// Replacer le sommet $s$ dans la liste ouverte au bon endroit,
// en tenant compte de la nouvelle valeur de $F(s)$.
// le même sommet ne doit pas se trouver 2 fois dans la liste.
$LO \leftarrow LO - \{s\}$ ▷ Solution simple mais peu efficace
$F(s) \leftarrow G(s) + H(s, a)$
$LO \leftarrow LO \cup \{s\}$
**end if**
**end if**
**end for**
**end if**
**end while**

L'algorithme se termine bien évidemment lorsque l'on atteint le sommet voulu. Pour retrouver le plus court chemin, il suffit, en partant de $a$, de remonter de père en père jusqu'à $d$.

**Remarque1 :** la justesse du chemin dépend beaucoup de la fonction heuristique utilisée. Dans notre cas, on peut utiliser $H(P_1, P_2) = (|P_1.x - P_2.x| + |\ P_1.y - P_2.y|)\ /5$

**Remarque2 :** on peut arrêter l'algorithme dans 2 cas :

1. Soit le sommet $a$ est extrait de la liste ouverte, auquel cas on a trouvé le plus court chemin. C'est le cas de la version présentée.
2. Soit le sommet $a$ est ajouté dans la liste ouverte, auquel cas on a trouvé un BON chemin pour aller de $d$ à $a$, mais il n'est pas sûr que ce soit le plus court.

**Remarque3 :** le terme « liste » ne désigne pas nécessairement le type abstrait « liste chaînée » vu en cours. La "liste fermée" ne requiert pas nécessairement une structure de liste chaînée : vous avez juste besoin de savoir si un sommet est ou non dans cette "liste fermée". Un champ supplémentaire dans la structure de sommets indiquant l'appartenance ou non à la liste fermée peut être simplement utilisé. La liste ouverte a pour rôle essentiel de faciliter la recherche du sommet ayant la plus petite valeur $F(s)$ parmi les sommets atteignables. Il est donc efficace de l'implanter selon un tas.

**Remarque4 :** il est possible d'améliorer la formulation de l'algorithme en pseudo-code. Par exemple, les étapes de mise à jour de la liste ouverte peuvent être factorisées...

# 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 :

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

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

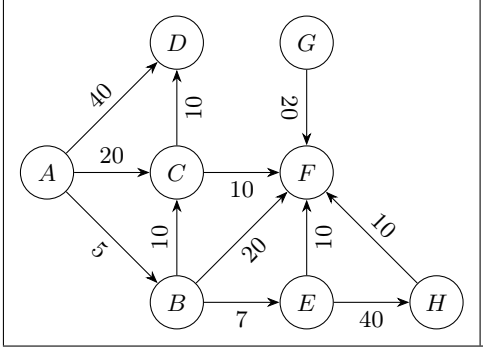
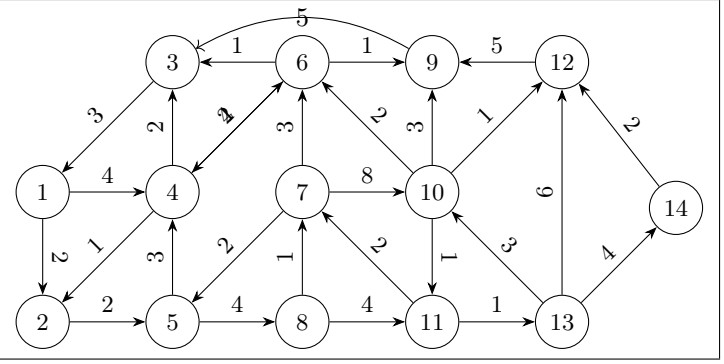
— NBS lignes dont le format est le suivant :

1. un entier : numéro de la station ;
2. deux réels indiquant la latitude et la longitude
3. une chaîne de caractères (sans séparateurs) contenant le « nom de la ligne » (par exemple M1, M3bis, T3, A1 pour le fichier métro parisien)
4. une chaîne de caractères contenant le nom du nœud (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 dans le répertoire /users/prog1a/C/librairie/projetS22019.

— `graphe1.txt` et `graphe2.txt` contiennent les petits graphes ci dessus

— `metroetu.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 le réseau routier américain. Ne pas recopier ces fichiers dans vos répertoires, car il sont très volumineux et comportent de 250000 à 14000000 sommets. Ils permettent de voir 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 `FILE` à notre fichier réel à l'aide de la fonction `fopen`. On peut ensuite récupérer 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. 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 nom de la station (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.

```
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 dans ce
   cas */
fgets(mot,511,f);
if (mot[strlen(mot)-1]<32) mot[strlen(mot)-1]=0;
/* mot contient 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 par */
fgets(mot,511,f);

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

## 5 Travail demandé

Le travail 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

### 5.1 Réseau routier américain

L'objectif est donc de construire le graphe 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 d'afficher le chemin trouvé ensuite. Il y a bien sur des sens uniques et 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.

Vous pouvez obtenir une solution en utilisant l'exécutable pcc par la commande :

```
/users/prog1a/C/librairie/projetS22019/pcc grapheNewYork.csv
```

et la visualisation du graphe en utilisant l'exécutable pccgraph par la commande :

```
/users/prog1a/C/librairie/projetS22019/pccgraph grapheNewYork.csv 1
```

Commencez par travailler sur un graphe de petite taille `graphe1.txt` et `graphe2.txt`

Tous les fichiers sont dans le répertoire `/users/prog1a/C/librairie/projetS22019`. Ne copiez pas les fichiers dans votre espace de travail car ils sont volumineux.

### 5.2 Metro et RER Parisien

Le graphe est ici un peu particulier : il y a une station différente par ligne, même si deux lignes ont le même nom de station. Par exemple, on trouve une station 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). Si vous partez de Gare du Nord, il faut donc soit calculer le plus court chemin à partir de chacun de ces sommets ou bien définir un arc de coût nul entre chacun de ces sommets ou toute autre solution à votre goût.

La notion de correspondance est prise en compte : 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. La correspondance est modélisée un arc de coût pré-établi, de valeur 360 entre ces deux sommets.

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 de rentrer le nom (et non le numéro) de la station. Il faut faire alors 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. Vous pouvez alors créer des arcs de valeur nulle entre ces sommets.

## 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 : rôle de chaque module (couple de fichiers .c/.h)
- 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.
- 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. Bien répartir le travail pour permettre au projet de se dérouler complètement.
- 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.
- Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure. Ne pas écrire tout le code et compiler au dernier moment.
- Validez le programme réalisé sur les graphes simples avant de le tester sur des graphes plus conséquents.
- **Attention : vous devez vous assurer que la compilation et vos programmes fonctionnent sur les machines de l'école.**

## 6.3 Rendu final

Chaque membre du binome copiera l'ensemble des fichiers constituant le livrable dans les répertoires : /users/phelma/phelma2018/mon-login/tdinfo/seance15. Le livrable sera constitué :

- du rapport du projet, format PDF. N'oubliez pas de faire figurer vos noms...
- des sources de votre programme et du Makefile
- d'un fichier README expliquant comment compiler et lancer votre (vos) programme(s)

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
   (c) Exemple d'exécution
   (d) Les optimisations et les extensions réalisées
2. Suivi
   (a) Problèmes rencontrés
   (b) Planning effectif
   (c) Qu'avons nous appris et que faudrait il de plus ?
3. Conclusion

# 7 Extensions

## 7.1 Optimisation

### 7.1.1 Recherche du minimum dans la liste ouverte

La recherche du minimum dans la liste ouverte peut se révéler coûteuse, car elle est en o(n) dans une approche classique. Si on implante la liste ouverte sous forme d'un tas, le maintien du tas est en o(log(n)) et le minimum est au sommet de ce tas : sa recherche est donc en o(log(n)) car la suppression du minimum oblige ensuite à maintenir la structure du tas (voir le TD sur les tas).

La principale difficulté ici est que la valeur $F(s)$ des sommets déjà dans le tas peut être modifiée par la suite (Etape de l'algorithme : replacer le sommet s dans la liste ouverte au bon endroit). Dans ce cas, il faut aussi mettre à jour le tas, car le sommet modifié peut éventuellement ne plus respecter la condition sur les tas : le père est plus petit que les 2 fils. Il faut donc retrouver un sommet dans le tas et éventuellement faire remonter le sommet dont le $F(s)$ est modifié. Comme la structure de tas n'est pas faite pour facilement trouver un élément, vous pouvez définir un champ supplémentaire dans la structure de sommet, qui contiendra l'indice du tableau tas où se trouve le sommet ou -1 si le sommet n'est pas dans le tas. Retrouver un sommet est alors de complexité o(1).

### 7.1.2 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 la recherche rapide. Notez que 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.