Projet d'informatique : 1A PET

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.

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 prototypes de fonctions, 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 des données du graphe avant même d'essayer de trouver le plus court chemin.
  - Tests d'intégration : quels sont les tests que vous allez faire pour prouver que l'application fonctionne, sur quels exemples.
- les modules (couples de fichiers .c et .h), les fonctions contenues dans chaque fichier
- 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

Ce document est essentiel et doit permettre ensuite de coder rapidement votre application en vous répartissant les tâches.

Séances suivantes :
Ces séances servent à débugger et à poser des questions aux enseignants. Si la répartition est bien faite, chaque membre du binôme peut avancer sa partie de travail.

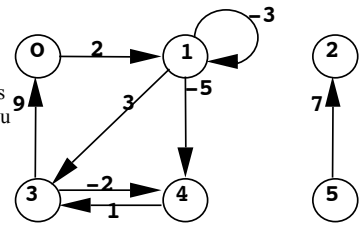
Dernière séance :
Il faut effectivement finir de réaliser l'intégration et regrouper les codes des 2 membres du binôme. Vous copierez l'ensemble des fichiers sources dans le répertoire /users/phelma/phelma2011/tdinfo/mon-login/seance14.
Vous ferez un **rapport** contenant l'analyse initiale, les tests prévus, la répartition et le planning effectif. Il indiquera l'état final du programme réalisé, ce qui fonctionne, ce qui ne fonctionne pas, expliquant les différences entre l'analyse faite à la première séance et une impression des tests que vous aurez fait. Le rapport fera au plus 20 pages, non compris le code, dont voici un plan indicatif:

1- Intro
2- Spécifications
  2.1 Données : description des structures de données
  2.2 Fonctions : Entete et rôle des fonctions essentielles
  2.3 Tests : quels sont les tests prévus
  2.4 Répartition du travail et planning prévu : qui fait quoi et quand ?
3- Implémentation
  3.1 Etat du logiciel : ce qui fonctionne, ce qui ne fonctionne pas
  3.2 Tests effectués
  3.3 Exemple d'exécution
  3.4 Les optimisations et les extensions réalisées
4- Suivi
  4.1 Problèmes rencontrés
  4.2 Planning effectif
  4.3 Qu'avons nous appris et que faudrait il de plus?
  4.4 Suggestion d'améliorations du projet
5- Conclusion

Projet 2012 : Plus court chemin dans un graphe

L'objectif du projet est de calculer le meilleur itinéraire, i.e. le plus court chemin entre 2 stations de métro parisien. Un fichier vous donne l'ensemble des stations et des relations entre 2 stations. Le métro se représente facilement par un graphe.

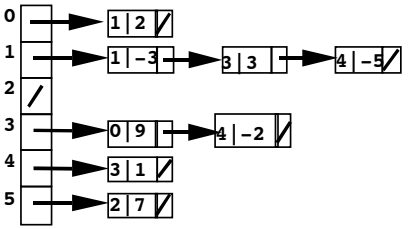
Définition & Terminologie
Un Graphe est défini par un couple G[X,A] où X est un ensemble de nœuds ou sommets et A est l'ensemble des paires de sommets reliés entre eux (arêtes du graphe ou « arc »)



- Arc = arête orientée
- **chemin = séquence d'arcs menant d'un sommet i à un sommet j**
- 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 des sommets sont ceux qui sont reliés à ce sommet par un arc.

Représentation
On peut représenter un graphe en numérotant les sommets et les arcs par des entiers. Un arc est un triplet "sommet de départ","sommet d'arrivée","coût de l'arc".

Matrice d'incidence sommets-arcs par liste chaînées
On utilise un tableau de listes chaînées pour représenter les arcs entre sommets. L'indice du tableau est le numéro du sommet et donne accès à la liste des arcs qui partent de ce sommet et qui mènent à un successeur. Chaque élément de la liste contient le numéro du successeur (sommet d'arrivée de l'arc) et sa valeur (son coût).



Exemple : les sommets sont numérotés de 0 à 5. Le tableau représentant le graphe contient les listes des arcs, chaque arc contenant le sommet d'arrivée et le coût de cet arc. A gauche, le graphe sans les coûts sur les arcs, à droite, le graphe avec les coûts des arcs.

Plus court chemin dans un graphe

Dans un problème de plus court chemin, on considère un graphe orienté **G=(X, A).** Chaque arc $\mathbf{a_i}$ est muni d'un coût $\mathbf{p_i}$. Un chemin $\mathbf{C=< a_1, a_2, ..., a_n >}$ possède un coût qui est la somme des coûts des arcs qui constituent le 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 solutions dans ce cas.
- L'algorithme A* ou Astar qui utilise une heuristique pour trouver rapidement une solution. Il ne s'applique qu'aux graphes dont les arcs sont à valuation positive. De plus, il nécessite qu'il soit possible d'estimer la « distance » entre deux nœuds par une fonction heuristique.

AStar

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, ie 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 donc de trouver le plus court chemin entre un sommet départ **d** et un sommet arrivée **a**. Pour cela, on va "tracer" progressivement un chemin en parcourant les sommets en essayant à chaque étape de s'approcher au mieux de **a**.

Lorsque qu'on se trouve sur un sommet **s**, l'algorithme A* choisit le sommet **k** (voisin de **s)** qui nous « approche le plus » du sommet **a**. Pour cela, si on traite le sommet **s**, on sépare le cout C(s) du chemin entre **a** et **d** et passant par **s** en 2 parties : C(s)= A(s) + H(s).

- A(s) : cout (d,s) : représente le cout du chemin le plus court entre **d** et **s**. Il est connu quand on traite le sommet **s** puisque **s** a déjà été atteint.
- H(s,a) estime le cout du chemin entre **s** et **a.** Le cout réel du plus court chemin entre **s** et **a** est inconnu. On utilise une heuristique, minorant le chemin réel. Par exemple, si les sommets **s** et **a** sont de coordonnées x_1,y_1 et x_2,y_2, la fonction H peut être la distance euclidienne (ligne droite) ou la fonction $(abs(x_1-x_2)+abs(y_1-y_2))/2.$, qui minore la distance euclidienne.

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

Tout au long du calcul, on va utiliser et maintenir deux listes :

- **La liste ouverte**, qui contient les sommets du graphe dont un voisin a déjà été atteint et traité. Ces sommets sont donc accessibles depuis **d**. **Cette liste sera triée en fonction de la valeur C(s) des sommets la composant**.
- **La liste fermée**, qui contient les sommets du graphe qui ont été atteints, traités et dont le plus court chemin depuis **d** est donc connu.

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

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

// Initialisation

- Calculer les valeurs A(**d**)=0, H(**d,a**) et C(d)=H(**d,a**).
- Tous les autres sommets **k** ont un C(**k**), un A(k) et un H(d) initialisés à « infini ».
- La **liste ouverte** est initialisée avec le sommet **{d}.**
- La **liste fermée** est vide

// Algorithme itératif

- Tant que la **liste ouverte** n'est pas vide ET que l'arrivée n'est pas atteinte
  - Extraire le sommet **k** de valeur C(**k**) la plus faible de cette liste
  - Si **k==a**, on a atteint l'arrivée et on a trouvé le plus court chemin
  - Sinon
    - // **k** est considéré comme le sommet atteignable le plus intéressant
    - // car rapprochant le plus de **a** (au sens de l'heuristique choisie).
    - // On va donc avancer d'un sommet, en considérant qu'on passe par **k**
      - On met le sommet **k** dans la liste fermée
        - // k est atteint, la valeur A(**k**) est le plus court chemin entre **d** et **k**
        - // maintenant, il faut mettre à jour les nouveaux sommets atteignables à partir de ce
        - // sommet k et modifier la liste ouverte
      - Pour tous les sommets **s** voisins de **k** qui ne sont pas dans la liste fermée
        - Si le sommet **s** n'est pas dans la liste ouverte
          - // Il n'a jamais été considéré
          - // Ajouter ce sommet à la liste ouverte.
          - // Il est possible qu'il soit le plus court pour aller de d à s
          - Initialiser Pere(s) à **k**
          - Calculer les valeurs de A(s)=A(k)+cout(k,s)
          - Calculer H(s,a) et C(s)
          - Insérer le sommet dans la liste ouverte au bon endroit
        - Sinon
          - // Comme **s** est dans la liste ouverte, on a déjà calculé un A(s)
          - si A(k)+cout(k,s) < A(**s**)
            - // On vient donc de trouver un nouveau chemin de **d** à **s**,
            - // plus court que celui trouvé précédemment
            - Mettre à jour Pere(s) à **k** pour tenir compte du nouveau chemin
            - Mettre à jour la valeur de A(s) = A(k)+cout(k,s)
            - Mettre à jour C(s)
            - Replacer le sommet s dans la liste ouverte au bon endroit, en tenant compte de la nouvelle valeur de C(s). Attention : le même sommet ne doit pas se trouver 2 fois dans la liste.

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

Remarque1 : on peut arrêter l'algorithme dans 2 cas :

- Soit le sommet a est ajouté dans la liste fermée, auquel cas on a trouvé le plus court chemin.

- 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 chemin. C'est la qualité de la fonction H qui permet d'obtenir rapidement ce chemin.

Remarque2 : Le terme « liste » ne désigne pas nécessairement le type abstrait « liste chainée » vu en cours. La "liste fermée" ne requiert pas nécessairement une structure de liste chainée : vous avez juste besoin de savoir si un sommet est ou non dans cette "liste fermée". La liste ouverte a pour rôle essentiel est de retrouver facilement le sommet ayant la plus petite valeur C(s) parmi les sommets atteignables. Il est donc possible et efficace de l'implanter par un tas.

Remarque3 : l'algorithme en pseudo-code a été écrit pour être le plus clair possible. En l'analysant, vous pourrez vous apercevoir qu'il est possible de le formuler de manière plus optimale. Par exemple, les étapes de mise à jour de la liste ouverte peuvent être factorisées…

Format des fichiers de données

Pour tester votre algorithme, vous disposez de trois fichiers de données avec des graphes de différente taille : un petit (graphe1_2012.csv), un très grand (graphe3_2012.csv), et un troisième qui encode le réseau métro/rer/tram parisien (metro2012.csv),. Dans ces fichiers, les coordonnées des sommets sont des coordonnées GPS comprises dans le rectangle (2.2 ; 48.2) (3.0 ; 49.1).

Le format de ces fichiers est le suivant :

Première ligne : deux entiers ; nombre de sommets (X) nombre d'arcs (Y)

Deuxième ligne : la chaîne de caractère "Sommets du graphe"

X lignes dont le format est le suivant :

- un entier : numéro de la station ;
- deux réels indiquant la latitude et la longitude
- une chaine de caractères (sans séparateurs) contenant le « nom de la ligne » (par exemple M1, M3bis, T3, A1 pour le fichier métro parisien)
- et une chaîne de caractères contenant le nom du nœud (qui peut contenir des séparateurs, par exemple des espaces).

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

Y lignes dont le format est le suivant :

- un entier : numéro du sommet de départ
- un entier : numéro du sommet d'arrivée
- un réel : valeur ou cout de l'arc

Remarque importante pour la lecture de ces fichiers en C:

La lecture des lignes contenant les sommets du graphe (les X lignes) doit se faire en lisant d'abord un entier, deux réels et une chaine, puis une chaine 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 station (chaine avec séparateurs) avec la fonction fgets :

```
fscanf(f,"%d %lf %lf %s", &(numero), &(lat), &(longi), line);
fgets(mot,511,f);
if (mot[strlen(mot)-1]<32) mot[strlen(mot)-1]=0;
```

numéro contient alors l'entier, lat et longi la position, line le nom de la ligne et mot le nom du sommet.

Fichiers metro2012.csv: le métro, le tramway et le RER de Paris

Cet exemple est un graphe avec environ 700 sommets et 1900 arcs. Chaque station est un sommet du graphe. Attention, il y a une station différente par ligne, même si deux lignes ont la même station. La notion de correspondance est prise en compte : si deux lignes 1 et 2 se croisent

avec correspondance, deux sommets 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. Le fichier inclut aussi les correspondances à pied (exemple : entre Gare du Nord et La chapelle). Dans ce cas, le coût pré-établi est de 600. Le coût des autres arcs dépend du moyen de transport (métro, tramway et RER). Pour tous les arcs, on est assuré que le coût est toujours supérieur à :

*cout_min(s, a) = 40 + distance_euclidienne(s,a)*3900.*

(où 40 estime un temps d'arrêt en station et 1/3900 minore la vitesse du moyen de transport le plus rapide, c'est à dire du RER).

Cette remarque vaut incidemment pour les trois fichiers de donnée.

Remarques :

- Chaque arc est orienté, donc il y a par exemple un arc entre le sommet 1 et 12 et un arc entre le sommet 12 et 1, et ces deux arcs ne sont pas identiques.
- Plusieurs stations portent le même nom : lorsqu'une station se trouve sur 2 lignes différentes, elle est alors considérée comme 2 sommets distincts du graphe, mais ces 2 sommets portent le même nom. Par exemple, *Montparnasse Bienvenue* est une correspondance pour 4 lignes : les lignes 4, 6, 12 et 13. On trouve donc 4 sommets pour *Montparnasse bienvenue*, les sommets 98, 138, 300, 361 correspondant à ces lignes, les 16 arcs (*Saint Placide-Montparnasse-Vavin*, *Pasteur-Montparnasse-Quinet*, *Notre Dame des Champs-Montparnasse-Falguière*, *Duroc-Montparnasse-Gaîté*), ainsi que des arcs reliant les différents nœuds *Montparnasse bienvenue.*

Heuristique de cout H(s,a)

L'algorithme A* nécessite que la fonction heuristique d'estimation du coût entre deux sommets H(s,a) minore toujours le coût du plus court chemin entre s et a. De plus, l'efficacité de l'algorithme A* dépend de la qualité de H(s,a) : plus l'évaluation est proche du coût réel du meilleur chemin, meilleure sera la convergence ; mais il convient de limiter la complexité de la fonction heuristique....

Pour définir la fonction H(s,a), il faut donc avoir une idée des coûts des arcs du graphe. Sur la base du paragraphe précédent, il apparaît qu'on peut choisir :

*H(s, a) = 40 + distance_euclidienne(s,a)*3900.*

Travail demandé

Faire une application qui demande les numéros des stations de départ et d'arrivée et qui calcule et affiche le plus court chemin pour ce voyage.

Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure.

Ayez recours aux 3 fichiers de données pour tester votre logiciel.

Dans un premier temps, vous désignerez les stations en utilisant leur numéro (un entier). Vous validerez le programme réalisé sur les graphes simples avant de le tester sur le métro. 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 chaine de caractères) à **tous** les sommets de même nom.

Attention pour les stations de départ et d'arrivée, car une station sur plusieurs lignes est un sommet différent. Pour partir de Montparnasse bienvenue, on peut utiliser les sommets 98, 138, 300, 361. De même pour l'arrivée. Et vous ne savez pas à priori quelle est la ligne que vous allez prendre au départ et celle par laquelle vous arrivez.

Ne vous perdez pas et Bon voyage !!!

Quelques éléments pris en compte dans la notation

Rapport : 3 points
Lecture du graphe : 3 points
Calcul du plus court chemin: 3 points
Affichage du chemin sommet par sommet : 3 points
Gestion des noms des stations : 3 points
Tests : 4 points
Code commenté : 1 point
Extensions : graphisme, optimisation, etc : bonus
Plagiat interne ou externe à phelma : 0/20