PROJET : LE TOUR DU MONDE LE MOINS CHER

Ce projet sera réalisé en binôme, sur les 4 dernières séances d'informatique ET en dehors des séances d'informatique.

Attention : La réussite de ce projet exige du travail en dehors des séances.
Pour la première séance, vous devez avoir lu le sujet, préparé des questions et formé les binômes.

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, 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.
- Le planning de réalisation du projet jusqu'à la date du rendu.

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

1 Le problème du voyageur de commerce

L'objectif du projet est faire le tour du monde le moins cher. C'est le « problème du voyageur de commerce », qui doit visiter N villes en passant par chaque ville exactement une fois. Il commence par une ville quelconque et termine en retournant à la ville de départ. On connaît le coût (temps, distance, ou autre chose...) des (N*(N-1)/2) trajets entre ces N villes. Quel circuit faut-il choisir afin de minimiser le coût du parcours ?
Si l'on considère que toutes les villes sont reliées entre elles, on peut représenter l'ensemble des connexions possibles par un *graphe non orienté, complet, pondéré* (cf. Figure 1). Un graphe est un ensemble de sommets reliés par des arêtes. Par exemple, les arbres sont des graphes mais dont les sommets ont la contrainte de n'avoir qu'un seul parent. Un graphe est complet si tous les sommets sont reliés entre eux 2 à 2. Un graphe est non orienté s'il n'y a pas de sens pour aller d'un sommet à un autre (p.ex.: on peut prendre l'avion pour aller de Paris à Rome et vice-versa). Un graphe est pondéré s'il le fait d'emprunter une arête a un coût.
Formellement, le problème du voyageur devient : partant d'un *sommet* donné, il s'agit de trouver le *circuit hamiltonien* de *moindre coût* passant par tous les *sommets* d'un *graphe non orienté, complet, pondéré.* Dans un graphe, un cycle hamiltonien est un chemin d'arêtes passant par tous les sommets une fois et revenant au nœud de départ (cf. Figure 1).
Le voyageur de commerce est un problème NP-complet : la solution exacte est de complexité N! . Il n'existe pas d'algorithme de complexité polynomiale connu. Il est donc impossible de trouver l'optimum en un temps raisonnable lorsque le nombre de « villes » devient grand (plus de 85000) ! Au delà, il faut utiliser des algorithmes d'approximation utilisant des heuristiques, qui aboutissent à une solution « pas trop mauvaise » en un temps acceptable.

Figure 1 : **Gauche: Un voyageur cherchant son cycle hamiltonien; droite: exemple de cycle hamiltonien**

2 Résolution du problème par la méthode de l'arbre couvrant minimum

Parmi les méthodes approchées, ce projet s'intéresse à celle de **l'arbre couvrant minimum.** L'arbre couvrant minimum d'un graphe est une partie du graphe (un sous-ensemble de ses arêtes) qui passe par tous les sommets du graphe une et une seule fois, et dont le coût est minimum. Dans le cas d'un graphe complet le parcours préfixé d'un arbre couvrant minimal du graphe est une solution approchée du voyageur de commerce. On peut montrer que le circuit ainsi obtenu a un coût au plus double du coût optimal. Un exemple de résolution par arbre couvrant minimum est illustré Figure 2.

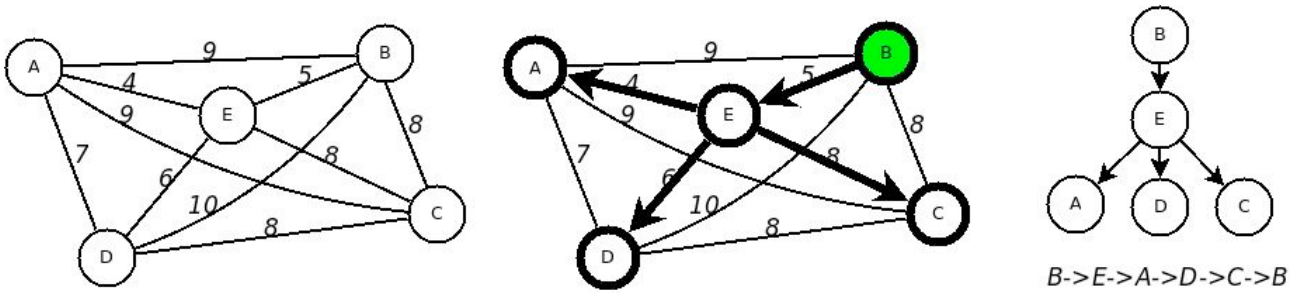
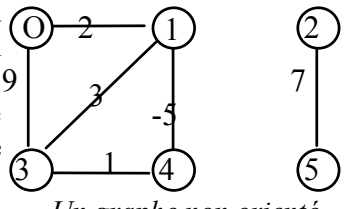


Figure 2 : **Résolution du problème du voyageur de commerce par l'arbre couvrant minimum**

Dans un premier temps, partant d'un sommet quelconque (ici **B**), on détermine l'ensemble des arêtes qui forme l'arbre couvrant minimum en utilisant *l'algorithme de PRIM*.

Dans un second temps, à partir de cette liste d'arêtes, on reconstruit l'arbre couvrant minimum explicite, c'est à dire une structure arborescente dont chaque nœud contient un sommet du graphe et qui respecte la connectivité des sommets de l'arbre couvrant minimum (cf. schéma du centre de la Figure 2).

Dans un troisième temps, on parcourt cet arbre (parcours préfixé) pour en extraire une file de sommets (de villes) ; cette file sera la solution approchée du problème du voyageur de commerce (cf. schéma de droite de la Figure 2)

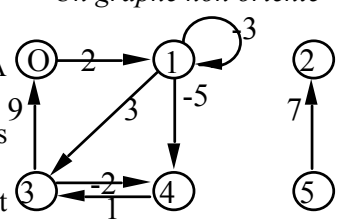


Un graphe non orienté

2.1 Définition & Terminologie

Un Graphe est défini par un couple G[X,A] où X est un ensemble de sommets et A est l'ensemble des paires de sommets reliés entre eux.

- Arc et Graphe orienté : chaque lien ou arc entre 2 sommets est à sens unique.
- Arête et Graphe non orienté : les liens ou arêtes entre 2 sommets ne sont pas orientés.



Un graphe orienté

- Chemin: séquence d'arcs/arêtes 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/arête et, par extension, à un chemin.
- voisins d'un sommet: les sommets qui sont reliés à ce sommet par un arc/arête.
- Graphe connexe : il existe au moins un chemin entre tous les sommets.
- Graphe complet: tous les sommets sont voisins de tous les autres sommets. Un graphe complet est toujours connexe.

3 Ensemble des arcs formant l'Arbre Couvrant Minimum

L'arbre couvrant minimum d'un graphe est un arbre dont chaque nœud contient un sommet du graphe, qui passe par tous les sommets du graphe, qui respecte la connectivité des sommets et dont le coût (somme des coûts des arêtes) est minimum. Il permet d'optimiser la longueur de câble nécessaire pour relier différents points d'un réseau informatique par exemple. Il est aussi utilisé pour la connexion de réseaux locaux au niveau 2 des couches ISO par des ponts ou bridges. Il peut bien sûr exister plusieurs arbres couvrant minimum pour un même graphe. La notion d'arbre couvrant minimum est définie pour les graphes *non-orientés*. Elle s'applique aussi bien à des graphes *complets* et des graphes *non complets*. Pour les graphes non *connexes*, elle s'entend pour chacune des parties connexes du graphe.

Un arbre couvrant minimum n'est pas nécessairement représenté au moyen du type abstrait « arbre ». Il est défini par l'ensemble des arêtes (des arcs[1]) du graphe qui le constituent, c'est à dire l'ensemble des arêtes connexes qui relient tous les points du graphe, par lesquels on ne passe qu'une seule fois et dont la somme des coûts est minimum. La première étape du projet consiste donc à trouver l'ensemble des arcs formant l'arbre couvrant minimum.

3.1 Algorithme de PRIM

On considère un graphe **G=(X, A).** Chaque arête $\mathbf{a_i}$ est munie d'un poids $\mathbf{p_i}$. On cherche l'ensemble des arêtes **eACM=< a_1, a_2, ..., a_n >** formant un arbre couvrant minimum. En partant d'un sommet choisi arbitrairement, l'algorithme de Prim fait croitre l'arbre à partir de ce sommet. Cet algorithme est incrémental. À chaque étape, on ajoute à l'ensemble d'arêtes **eACM** une nouvelle arête : celle qui relie un sommet pas encore dans l'arbre à un sommet déjà dans l'arbre et qui a le plus faible coût. Cela revient à ajouter à l'arbre le sommet du graphe qui pas encore dans l'arbre et qui est« le plus proche » de l'arbre.

Cet algorithme s'exprime facilement de la façon suivante, en se limitant aux **graphes connexes**.

On considère ***eACM=< a_1, a_2, ..., a_n >,*** l'ensemble d'arêtes représentant l'arbre minimum en cours de construction. Cette liste permet de déterminer l'ensemble des sommets « déjà atteints ».

On considère un sommet de départ **d** quelconque ;

Tant que tous les sommets du graphe n'ont pas été atteints :

Parcourir toutes les arêtes qui lient les sommets déjà atteints à un sommet pas encore atteint

*Sélectionner l'arête **a** qui a le plus petit coût*

*Ajouter cette arête **a** à **eACM** ;*

Fin tant que

3.2 Structures de données utilisées

Pour implanter cet algorithme, on maintient deux ensembles tout au long du calcul:

- **C :** ensemble des sommets du graphe atteignables depuis les sommets déjà atteints ; initialement **C={d}.** On pourra utiliser une simple liste pour stocker cet ensemble de sommets ou un tas (voir optimisations).
- **fileACM :** ensemble des arcs formant l'arbre couvrant minimum en cours de construction. Pour stocker l'ensemble, on choisira d'utiliser une *file* d'arcs. Ce choix permettra de faciliter l'étape de construction de l'arbre des nœuds, que nous verrons plus tard.

[1] Tous nos graphes sont non orientés. Mais, comme on utilise une structure de donnée « graphe orientés » pour représenter ces graphes non orientés (avec deux arcs, un dans chaque sens, pour chaque arête), on parlera plus loin « d'arc » et non pas « d'arêtes ».

Un arc est un triplet "sommet de départ","sommet d'arrivée","coût de l'arc". Pour représenter une arête, on utilisera deux arcs orientés, un dans chaque sens.

Il existe plusieurs modes de représentation d'un graphe afin de gérer les liens entre sommets. Nous utiliserons une liste d'adjacence des voisins d'un sommet : c'est la liste des arcs sortant de ce sommet. Cette liste est stockée dans le champ *voisins* d'un sommet.

Le graphe sera donc défini par un tableau de sommets. Chaque sommet contient ses propres caractéristiques ainsi que des informations utiles à l'algorithme :

Champs propres au sommet, qui définissent le graphe :

- **numéro :** le numéro du sommet
- **nom :** le nom du sommet
- **x, y :** positions du sommet (pour la représentation graphique du graphe)
- **voisins :** la liste d'adjacence, liste de pointeurs vers les arcs sortant de ce sommet

Champs additionnels propres à l'algorithme de PRIM :

- **arrive_par :** pointeur vers le meilleur arc par lequel le sommet pourrait être connecté à l'ACM en cours de construction.
- **PPC** : coût (minimum) de l'arc qui connecte ce sommet à l'ACM en cours. Cet arc est stocké dans le champ arrive_par.

3.3 Algorithme : file d'arcs de l'Arbre couvrant minimum partant du sommet d

début

// le cout des sommets est infini au début

pour tous les sommets i de G **faire**

// les sommets ne sont pas encore dans l'ACM

$PPC_i \leftarrow +\infty$

$arrive\_par_i \leftarrow NULL$

fin pour

// le cout du sommet de départ est nul car il est dans l'ACM

$PPC_d \leftarrow 0$

$arrive\_par_i \leftarrow NULL$

// l'ACM est vide au départ et le sommet d est atteignable

$fileACM \leftarrow \{\}$

$C \leftarrow \{d\}$

Faire // On cherche le sommet non présent dans l'ACM

// qu'il est le moins coûteux d'atteindre

Sélectionner le sommet j de C de plus petite valeur PPC_j

// supprimer j de l'ensemble C

$C \leftarrow C \setminus \{j\}$

// ajouter l'arc par laquelle j est connecté à l'ACM, sauf pour le sommet de depart

Si j n'est pas d

$fileACM \leftarrow fileACM \cup \{arrive\_par_j\}$

Fin si

/* Un nouveau sommet est inclus dans l'ACM. Il faut mettre à jour les couts de tous les sommets qui lui sont adjacents ainsi que les liens arrive_par compte tenu du nouveau cout de connexion de ces sommets avec l'ACM en passant par j. Attention, il existe peut être d'autres connexions possibles.*/

pour tous les sommets k adjacents à j **faire**

/*S'il est plus court de passer par j pour connecter le sommet k à l'ACM, on met le sommet k à jour : c(j,k) est le coût de l'arc entre j et k */

si $PPC_k > c(j,k)$)

alors

// On a trouvé une « meilleure arête » pour connecter le

// sommet à l'ACM en cours de construction, en passant par j

// Nouveau cout de la connexion de k à l'ACM passant par j

```
                    PPC_k ← c(j,k) ;
                    arrive_par_k=j;
                    si le sommet k n'est pas dans C
                        // on peut désormais atteindre le sommet k
                        // depuis l'ACM en cours de construction
                    alors
                        C ← C ∪ {k}
                    sinon
                        Mettre C à jour si besoin (tas, liste triée)
                    fin si
                fin si
            fin pour
        tant que C n'est pas vide
        Retourner fileACM , la file des arcs inclus dans l'ACM
fin
```

Un exemple de déroulement de cet algorithme est disponible sur le site TD info.

4 Construction explicite de l'ACM

L'algorithme précédent construit la file des arcs formant l'ACM. On calcule le coût de l'ACM en parcourant cette file et en sommant les couts des arcs. De même, l'affichage de l'ACM (graphique ou textuel) peut simplement se faire en affichant chaque arc.

On va maintenant reconstruire l'arbre explicite correspondant à l'ACM, c'est à dire l'arbre dont chaque nœud contient un sommet et a pour fils les sommets reliés au nœud précédent par un arc de l'ACM. C'est un arbre n-aire, que nous représenterons par un arbre fils-frere (voir cours). On pourrait aussi utiliser une liste de fils.

Pour construire l'arbre et le parcourir, on initialise l'arbre avec le sommet départ. Ensuite il faut parcourir les arcs de fileACM et ajouter chaque sommet d'arrivée de l'arc à notre arbre.

Soit arbreACM l'arbre explicite et fileACM la file obtenue par l'algorithme précédent, l'algorithme de construction peut se décrire par :

Créer la racine de l'arbre arbreACM et y mettre le sommet de départ.
***tant que** la file d'arc « fileACM » n'est pas vide*
    défiler un arc A de fileACM
    ajouter le sommet d'arrivée de l'arc A à l'arbre arbreACM
fin tantque
retourner arbreACM

Il est toujours possible d'ajouter le sommet d'arrivée de l'arc A à arbreACM. En effet, la file fileACM a été construite dans l'ordre de croissance de l'ACM. Elle est donc défilée dans l'ordre de croissance de l'arbre et on est sûr que le sommet de *départ* de l'arc défilé a déjà été traité et placé dans l'arbre.

Pour ajouter le sommet d'arrivée d'un arc à arbreACM, il faut commencer par trouver le nœud de l'arbre contenant le sommet de départ de l'arc et ensuite ajouter ce sommet d'arrivée aux fils de ce nœud. Une première solution naïve serait de parcourir les nœuds de arbreACM jusqu'à ce qu'on trouve ce nœud. Cette approche nécessiterait un algorithme récursif coûteux.

Pour accélérer la recherche du nœud, il faut le trouver à partir du sommet de départ de l'arc. Pour cela, on va **ajouter le champ arbreACM noeudArbreACM ; à la structure de donnée sommet du graphe,** dans lequel on conservera l'adresse du nœud de l'arbre qui contient le sommet. Chaque fois qu'un nouveau nœud est ajouté dans arbreACM pour un nouveau sommet, on stockera l'adresse de ce nœud dans le sommet. Lors de l'ajout d'un arc, retrouver le nœud contenant le sommet de départ de l'arc est immédiat. La fonction ajoutant le sommet arrivée d'un arc n'est plus récursive : elle est en (O(1)).La fonction d'ajout d'un sommet à l'arbre arbreACM devient :

// A : pointeur vers l'arc à ajouter. sommets : tableau des sommets
fonction : ajouter nœud arrivée de **arc A** à arbreACM
    sommetDepart ← ptr vers le sommet de départ de arc A
    sommetArrivee ← ptr vers le sommet d'arrivée de arc A

    noeudSommetDepart ← sommetDepart-> noeudArbreACM
        // noeudSommetDepart pointe le nœud de l'arbre ACM qui contient le sommet de.
        // départ. Il est toujours non NULL puisque ce nœud a déjà été ajouté à l'arbre
    Creer un nouveau nœud P;
    Mettre sommetArrivee dans P
        // P est un nouveau fils pour le sommet de départ : les autres fils de départ sont donc
        // ses freres.
    Mettre les fils de noeudSommetDepart comme freres de P
        // P est maintenant le premier fils du sommet de départ
    Mettre P comme fils de noeudSommetDepart
        // On indique au sommet où se trouve son nœud dasn l'arbre
    Mettre P dans le champ sommetArrivee-> noeudArbreACM
finFonction

5 Le problème du voyageur de commerce

Cette troisième étape affiche la solution approchée du voyageur de commerce à partir de l'arbre arbreACM explicite. Dans le cas d'un graphe complet (tous les sommets sont reliés entre eux 2 à 2), on peut montrer que le parcours préfixé d'un arbre couvrant minimal du graphe est un circuit de poids au plus double du poids de la solution optimale du « voyageur de commerce ». C'est ainsi que les 2 algorithmes précédents permettent d'approximer la résolution du problème du voyageur de commerce.

Un parcours en ordre préfixé est une fonction récursive. Il consiste à effectuer une action (ici, l'affichage) d'abord sur le nœud courant, puis appeler la fonction récursivement sur le fils, puis sur le frère. Vous implanterez cet algorithme et afficherez le circuit (la série de sommets) qui est la solution approchée du problème du voyageur de commerce.

6 Format des fichiers de données

Plusieurs graphes vous sont fournis pour tester votre programme. Le format de ces fichiers est le suivant :

Première ligne :
    deux entiers ; nombre de sommets (X) nombre d'arêtes (Y)
Deuxième ligne :
    une chaîne de caractère qui est : « Sommets du graphe »
X lignes :
    un entier, deux réels, et une chaîne de caractères : numéro du commet, coordonnées en x du sommet, coordonnées en y du sommet, nom du sommet
1 ligne :
    une chaîne de caractère qui est : « Arêtes du graphe : noeud1 noeud2 valeur »
Y lignes :
    un entier, un entier, un **réel** : sommet de départ, sommet d'arrivée, coût de l'arête

Remarque importante pour la lecture en C:

La lecture des lignes contenant les sommets du graphe (les X lignes) doit se faire en lisant d'abord un entier puis une chaine de caractères qui peut contenir des espaces. Pour cela, on utilise :

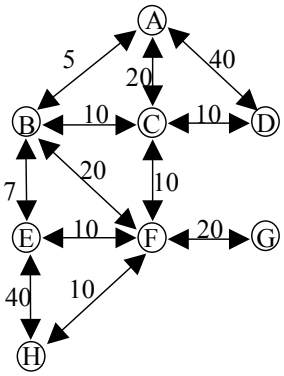
```
char mot[512] ;
fscanf(f,"%d %lf %lf %[^\n] ",&numero,&x,&y,mot);
```

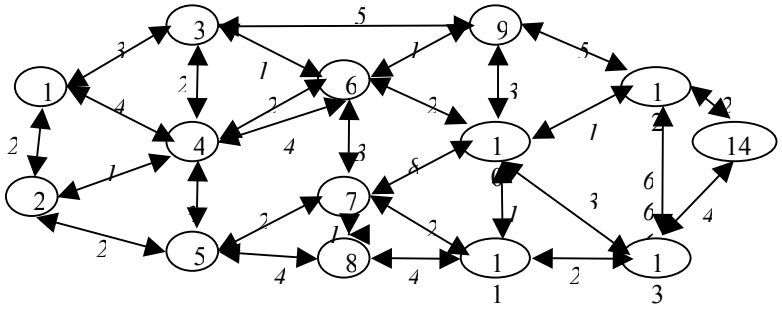
numéro contiendra alors l'entier, x et y les coordonnées et mot le nom du sommet.

6.1 Fichiers disponibles

Tous les graphes fournis sont *non-orientés*. Comme on utilise une structure de donne « graphe orientés » pour les stocker en mémoire, il faudra allouer deux arcs en mémoire pour chaque arête trouvée dans le fichier (un arc dans chaque sens).

Les fichiers graphe1.txt et graphe2.txt (figure ci dessous) sont des graphes *non complets* simples. Les fichiers graphe3.txt, graphe4.txt, graphe5.txt sont des graphes *non complets* plus importants (10000,100000,1000000 sommets). Ces 5 fichiers permettent de tester la construction de l'arbre couvrant (parties 2 et 3).





Pour la partie 4 (problème du voyageur de commerce), il faut que le graphe soit *complet*. Les fichiers graphe11.txt, graphe12.txt, graphe13.txt, graphe14.txt, graphe15.txt, graphe16.txt sont des graphes *complets* de complexité croissante (6, 10, 50, 100, 1000, 3000 sommets). Les fichiers monde.txt, france.txt sont également des graphes complets : monde.txt contient les capitales du monde, france.txt contient 300 villes françaises.

Attention : ne pas recopier dans vos répertoires les fichiers de données dont certains sont très volumineux.

7 Travail demandé

7.1 Réalisation

Vous allez réaliser 2 programmes :

1. un premier programme calcule juste les arcs appartenant à l'ACM ainsi que son cout (section 3). Il lit le graphe dans un fichier, affiche les arcs de l'arbre couvrant minimum et afficher la valeur du coût de cet arbre (§2). Ce programme est utilisable avec tous les fichiers fournis.
2. Un deuxième programme calcule le voyage optimal à effectuer à partir d'un sommet donné (section 4 et 5). Il lit le graphe dans un fichier, calcule les arcs de l'arbre couvrant minimum (§2) construit cet arbre (§3) et affiche le parcours préfixé de cet arbre (§4). Ce programme est utilisable avec les graphes complets uniquement.

Réaliser les améliorations et optimisations suivantes :

- Version graphique : en utilisant la SDL, afficher le graphe initial et son ACM..
- Algorithme de PRIM : le cout principal de l'algorithme de PRIM est du à la recherche du sommet pas encore dans l'ACM dont le cout est le plus faible. La version de base de la recherche de ce minimum est en o(nbSommet). La complexité totale est alors en o(nbArc*nbSommet). Pour optimiser, on peut utiliser un tas, dont la propriété est de contenir le minimum à la racine. Chercher le minima est alors en o(1). Par contre, si un des sommets k adjacents à celui de cout minimum est mis à jour par PRIM (etape $PCC_k \leftarrow$ c(j,k) ;) et s'il est déjà dans le tas, sa position dans le tas risque de ne plus être à jour. Il faut donc ajouter une mise à jour du tas dans ce cas (clause sinon du test si arete{j,k} n'est pas dans le tas). Comme le cout ne peut que avoir diminuer, il est possible qu'il faille remonter ce sommet dans le tas. Il faut encore trouver où est ce sommet dans le tas, et un champ supplémentaire contenant cette information dans la structure sommet sera utile. Avec un tas, la complexité est ici en log(nbSommet) pour la recherche du minimum, et majorée par o(nbAretes*log(nbSommet)) pour la complexité globale.
- En plus d'afficher le circuit solution approchée du problème du voyageur de commerce, calculez et affichez son coût. Attention : il faut tenir compte des arêtes liant les sommets qui ne sont pas en relation « père fils » dans l'arbre (lorsqu'on remonte dans l'arbre durant le parcours préfixé).
- Pour le parcours préfixé de l'ACM, recherchez et implantez une solution *itérative* en lieu et place de l'algorithme récursif. La solution itérative est plus coûteuse en mémoire, mais en général moins coûteuse en temps.

7.2 Conseils de développement

- Commencez par définir les structures de données dont vous aurez besoin pour représenter les sommets, les arcs, les listes, files dont vous avez besoin. Ecrivez et tester les fonctions de base sur ces types abstraits avant de passer à la suite.
- Prévoyez un développement incrémental en testant toutes vos fonctions au fur et à mesure.
- Attention : vous devez vous assurer que la compilation et vos programmes fonctionnent sur les machines de l'école.

7.3 Livrables

La date de rendu est vendredi 24 mai.
Vous copierez l'ensemble des fichiers constituant le livrable dans le répertoire :
/users/phelma/phelma2011/tdinfo/**mon-login**/seance14.
Le livrable sera a minima constitué :

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

Le rapport final fera au plus 20 pages, non compris le code.
Voici un plan indicatif:

1- Intro
2- Spécifications
    2.1 Données : description des structures de données
    2.2 Modules : rôles de chaque module (couple de fichiers .c et .h)
    2.3 Fonctions : prototype et rôle des fonctions essentielles
    2.4 Tests : quels sont les tests prévus
    2.5 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

7.4 Quelques éléments pris en compte dans la notation

Rapport : 3 points
Lecture du graphe : 3 points
Calcul de l'ACM : 3 points
Affichage texte de l'ACM dans l'ordre : 1 point
Construction de l'arbre: 3 points
Affichage texte préfixe : 2 points
Tests : 4 points
Code commenté : 1 point
Extensions : graphisme, optimisation, etc : bonus
Plagiat interne ou externe à phelma : 0/20