

ÉCOLE POLYTECHNIQUE
Promotion 2009, Année 2010-2011

INF431, Contrôle Classant 1
13 avril 2011

Philippe Baptiste, Frederic Magniez, Benjamin Werner

*Tous les documents du cours et les notes personnelles sont autorisés.
On attachera une grande importance à la concision, à la clarté, et à la
précision de la rédaction. Les trois parties sont largement
indépendantes.*

On traitera la partie 1 sur des copies jaunes et les parties 2 et 3 sur des copies roses.

Un voyageur de commerce doit visiter un ensemble donné de villes et revenir à son point d'origine. Le problème du voyageur de commerce (ou TSP pour Traveling Salesman Problem) consiste à trouver l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur.

Les domaines d'applications immédiats du TSP sont nombreux : problèmes de logistique, de transport, d'ordonnancement. Au delà, le TSP est un problème sous-jacent à de très nombreuses questions d'optimisation combinatoire.

Le TSP est difficile à résoudre et on ne connaît pas de méthode de résolution permettant d'obtenir des solutions optimales en un temps polynomial. Nous étudierons dans cette composition plusieurs variantes du TSP. A titre d'exemple, voici le tour le plus court des 15 plus grandes villes d'Allemagne (la distance utilisée dans cet exemple est la distance euclidienne).



PARTIE 1 : D'un graphe à l'autre

Dans cette première partie le problème est défini par un graphe $G = (S, A)$ non orienté et valué de n sommets. Chaque sommet représente une ville, et chaque arête $(i, j) \in A$ une route entre les villes i et j ; la valeur associée à (i, j) est la longueur de la route entre i et j , nécessairement strictement positive. Le graphe étant non orienté, si $(i, j) \in A$ alors $(j, i) \in A$; de plus les valeurs associées à (i, j) et (j, i) sont identiques. Intuitivement cela signifie que les routes (i, j) sont à double-sens, et que les distances entre les deux extrémités i et j sont les mêmes dans les deux sens.

Enfin précisons que G n'est pas un multigraphe : il y a au plus une arête entre i et j .

Attention : ce graphe n'est pas a priori complet. Pour mémoire un graphe est complet si et seulement si ses sommets sont tous reliés deux à deux.

L'objectif de cette première partie est de vérifier que le graphe est connexe et de construire une matrice des distances entre toutes les villes.

Question 1.1

Deux classes sont utilisées pour représenter le graphe G . La classe `Ville` représente les villes à l'aide des champs suivants :

- `nom` contient le nom de la ville,
- `id` est un identifiant numérique unique de la ville,
- `voisins` est la liste des routes dont l'une des deux extrémités est la ville en question.

La classe `Route` représente les routes entre les villes à l'aide des champs suivants :

- `a` et `b` décrivent deux villes,
- `distance` contient la longueur de la route entre `a` et `b`.

```
class Ville {
    String nom;
    LinkedList<Route> voisins;
    int id;
    // ...
}

class Route {
    Ville a;
    Ville b;
    int distance;
    // ...
}
```

On rappelle que la classe générique prédéfinie `LinkedList<E>` implémente l'interface `Collection<E>`. On redonne également quelques-unes de ses méthodes :

- `LinkedList()` : le constructeur qui fabrique une nouvelle liste vide
- `void add(E e)` : ajoute `e` en queue de liste
- `E removeFirst()` : renvoie le premier élément après l'avoir enlevé de la liste (et une exception si la liste est vide)

– boolean isEmpty() : renvoie true si et seulement si la liste est vide

Ecrivez les deux constructeurs Ville(String nom, int id) et Route(Ville a, Ville b, int distance). On sera attentif au fait qu'en créant une route entre a et b, les listes de voisins doivent être mises à jour. Lors de la création, on pourra supposer, sans le vérifier, que la route n'a pas déjà été créée et également que a et b sont distincts.

Question 1.2

Ajoutez à Ville une méthode Route getRoute(Ville b) telle que a.getRoute(b) renvoie la route entre a et b si elle existe, et null sinon.

Question 1.3

Dans la suite, le problème du voyageur de commerce sera représenté par la classe Tsp décrite ci-dessous. L'entier n représente le nombre total de villes et le tableau villes de taille n contient toutes ces villes. On supposera que l'identifiant ville[i].id de la ville ville[i] est exactement i. En particulier, à partir de maintenant, les identifiants id seront compris entre 0 et n-1 inclus.

```
class Tsp {
    static int n;
    static Ville[] villes;
    // ...
}
```

Ecrivez une méthode public static boolean connexe() de la classe Tsp qui vérifie que le graphe des villes est connexe. Vous pouvez, au besoin, ajouter d'autres champs, par exemple à la classe Ville.

Question 1.4

On suppose maintenant que le graphe est bien connexe. On veut calculer une fois pour toutes les longueurs des plus courts chemins entre toutes les villes. Ces distances seront stockées dans une matrice distance[][] de la classe Tsp. En particulier, on aura distance[i][i]=0, pour toute ville d'identifiant i.

```
class Tsp {
    static int n;
    static Ville[] villes;
    static int distance[][];
    // ...
}
```

a) Montrez que la matrice distance des plus courts chemins doit être symétrique ($distance[i][j] = distance[j][i]$, pour toutes villes i, j) et satisfaire l'inégalité triangulaire ($distance[i][j] \leq distance[i][k] + distance[k][j]$, pour toutes villes i, j, k).

b) Indiquez des algorithmes utilisables pour calculer cette matrice des distances ; précisez éventuellement leurs complexités.

c) En faisant un choix parmi ces algorithmes, écrivez une méthode `public static void calculDistances()` de la classe `Tsp` qui initialise la matrice `distance[][]` puis la remplit. Vous n'êtes pas obligés de choisir la méthode la plus efficace.

PARTIE 2 : Des heuristiques de résolution

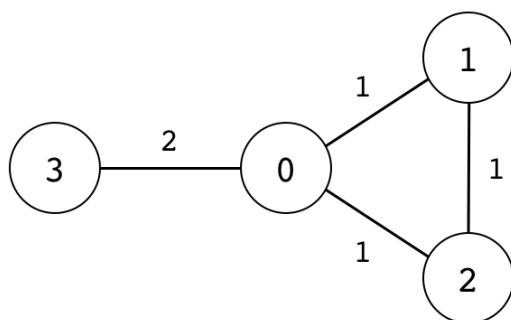
On s'intéresse maintenant au graphe complet valué donné par `distance[][]`. On cherche dans ce graphe complet une *tournée*, c'est-à-dire un cycle parcourant une et une seule fois chacune des villes avant de revenir à son point de départ.

On appelle *longueur* d'une tournée la somme des distances parcourues lors du parcours des villes incluant le retour au point de départ.

Question 2.1

On peut remarquer qu'une arête dans le graphe complet correspond à une ou plusieurs arêtes dans le graphe initial. Aussi, la tournée recherchée dans le graphe complet correspond à un cycle dans le graphe initial, ce dernier pouvant éventuellement passer plusieurs fois par une même ville.

On considère le graphe initial suivant :



Dessinez le graphe complet des distances correspondant. Donnez, pour ce graphe complet, une tournée de longueur minimale, en énumérant les sommets dans l'ordre.

Question 2.2

Dans la suite on supposera que la matrice des distances `distance[][]` a été calculée et tous les calculs se feront sur cette matrice (et non pas sur la représentation initiale du graphe).

Une tournée peut donc être vue comme une permutation circulaire des identifiants des villes. Ce cycle est représenté par un tableau `t` de `n` valeurs distinctes dans $\{0, 1, \dots, n-1\}$. Le cycle est donc $(t[0], t[1], \dots, t[n-1])$. Cette notation signifie que `t[0]` est l'indice de la première ville visitée, `t[1]` l'indice de la seconde et ainsi de suite jusqu'à `t[n-1]` qui est l'indice de la dernière ville visitée avant le retour en `t[0]`.

La classe `Tournee` représente une telle solution. Le constructeur ci-dessous initialise le cycle à `t[i]=i`. Il s'agit de la tournée qui consiste à visiter les villes par ordre croissant d'identifiant. Après la dernière ville visitée, d'identifiant `n-1`, la tournée revient à la première ville, d'identifiant `0`.

```
class Tournee {
    int[] t; // permutation cyclique
    Tournee() {
```

```

    t = new int[Tsp.n];
    for (int i = 0; i < Tsp.n ; i++)
        t[i] = i;
}
//...
}

```

On peut supposer que t encode toujours une tournée, mais a priori quelconque.

Ecrivez une méthode `public int longueur()` de la classe `Tournee` qui renvoie la longueur de la tournée.

Question 2.3

L'heuristique constructive la plus simple pour le TSP consiste à partir d'une ville puis d'aller de ville en ville, en visitant à chaque fois la ville non visitée la plus proche.

Donnez un exemple pour lequel cette heuristique ne renvoie pas la tournée la plus courte (on pourra éventuellement faire un dessin).

Ecrivez une méthode `public void plusProcheVoisin()` de la classe `Tournee` qui applique l'heuristique ci-dessus en partant de la ville $t[0]$. Quelle est sa complexité ?

Question 2.4

On définit un algorithme itératif qui améliore la tournée courante en supprimant deux arêtes non adjacentes de la tournée et en reconnectant les deux chemins résultants, puis recommence le cas échéant.



Ainsi dans l'exemple ci-dessus, les deux arêtes en gras ont été supprimées de la tournée de gauche pour construire la tournée de droite.

Ecrire une méthode `public boolean amelioration()` de la classe `Tournee` qui cherche les deux arêtes qui donnent un gain maximal lors de la transformation. Si ce gain est positif, la méthode renverra `true` après avoir effectué la modification sur la tournée. Si la tournée initiale ne peut être améliorée, celle-ci restera inchangée et la méthode `Tournee amelioration()` renverra `false`. On sera très attentif à la complexité de la méthode `Tournee amelioration()` qui devra être $O(n^2)$.

PARTIE 3 : Les tournées pyramidales

On reprend dans cette partie le contexte de la Partie 2 (lire si besoin son préambule).

Une tournée pyramidale de n villes $\{0, 1, \dots, n-1\}$ consiste à visiter une séquence de villes d'indices croissants jusqu'à la ville $n-1$, puis à visiter les villes restantes dans l'ordre décroissant des indices, et enfin à revenir à la première ville du cycle. Autrement dit, elle peut s'écrire de la manière suivante :

$$(t_1, t_2, \dots, t_p, t'_1, \dots, t'_{p'}),$$

avec $p + p' = n$ et $t_1 < t_2 < \dots < t_p = n-1$ et $n-1 > t'_1 > t'_2 > \dots > t'_{p'}$.

On rappelle que cette notation signifie que les villes sont visitées dans l'ordre $t_1, t_2, \dots, t_p, t'_1, \dots, t'_{p'}$, sans oublier le retour en t_1 .

Question 3.1

En utilisant le fait que la matrice distance est symétrique (distance[i][j] = distance[j][i], pour toutes villes i, j), expliquez pourquoi on peut supposer, sans perte de généralité, que $t'_1 = n-2$.

Question 3.2

On cherche à construire une formulation de programmation dynamique pour déterminer la meilleure tournée pyramidale. Pour ce faire, on introduit la notion de tournée j -pyramidale.

Soit $j \in \{2, \dots, n\}$ un entier donné. La permutation $(t_1, t_2, \dots, t_p, t'_1, \dots, t'_{p'})$ de $(0, 1, \dots, j-1)$ est une tournée j -pyramidale si et seulement si $p + p' = j$ et $t_1 < t_2 < \dots < t_p = j-1$ et $j-2 = t'_1 > t'_2 > \dots > t'_{p'}$.

On définit le *coût* d'une telle tournée j -pyramidale comme

$$\sum_{u=1}^{p-1} \text{distance}[t_u][t_{u+1}] + \sum_{u=1}^{p'-1} \text{distance}[t'_u][t'_{u+1}] + \text{distance}[t'_{p'}][t_1]$$

La *meilleure tournée j -pyramidale* recherchée sera désormais celle de coût minimal. Soit \mathcal{F}_j le coût de cette meilleure tournée j -pyramidale.

La définition du coût d'une tournée j -pyramidale est légèrement différente de celle de la longueur d'une tournée. Quelle est cette différence ? Comment calculer la longueur de la meilleure tournée pyramidale si on a calculé les meilleurs coûts $\mathcal{F}_2, \mathcal{F}_3, \dots, \mathcal{F}_n$?

Question 3.3

On définit pour $2 \leq j \leq n$:

$$A^j = \text{distance}[0][j-1] + \sum_{u=0}^{u=j-3} \text{distance}[u][u+1],$$

$$B_k^j = \mathcal{F}_k + \text{distance}[k-1][j-1] + \sum_{u=k}^{u=j-3} \text{distance}[u][u+1], \quad \text{pour } 2 \leq k \leq j-1.$$

Montrez que :

$$\begin{aligned}\mathcal{F}_2 &= \mathcal{A}^2 \text{ et} \\ \mathcal{F}_j &= \min(\mathcal{A}^j, \mathcal{B}_2^j, \mathcal{B}_3^j, \dots, \mathcal{B}_{j-1}^j), \text{ pour } j \geq 3.\end{aligned}$$

On expliquera en particulier ce que représentent les quantités \mathcal{A}^j et \mathcal{B}_k^j .

Question 3.4

Indiquez comment ces définitions peuvent se traduire en un programme dynamique. Quelle est sa complexité ?

Question 3.5

Il n'a, pour le moment, été question que du calcul du coût d'une tournée pyramidale optimale. Comment calculer explicitement une telle tournée pyramidale optimale ?