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.*

Solution.

```
Ville(String nom, int id) {
  this.nom=nom;
  this.id=id;
  voisins=new LinkedList<Route>();
}
```

```
Route(Ville a, Ville b, int distance) {
  this.a=a;
  this.b=b;
  this.distance=distance;
  a.voisins.add(this);
  b.voisins.add(this);
}
```

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.*

Solution.
Il vaut mieux utiliser `for` que `removeFirst` qui serait destructif.

```
Route getRoute(Ville b){
  for (Route r : this.voisins) {
    if (r.a.id == b.id || r.b.id == b.id) return r;
  }
  return null;
}
```

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`.

Solution.
Il suffit de faire une exploration en profondeur classique.

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

```
  public static void marquage(Ville v) {
    v.marque=true;
     LinkedList<Route> l=v.voisins;
    while (! l.isEmpty()) {
        Route r=l.removeFirst();
        Ville a=r.a, b=r.b;
        if (!a.marque) marquage(a);
        if (!b.marque) marquage(b);
    }
  }

  public static boolean connexe() {
     for (int i=0;i<n;i++)
       villes[i].marque=false;

     marquage(villes[0]);

     for (int i=0;i<n;i++)
       if (!villes[i].marque) return false;
     return true;
  }
```

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`*).*

Solution.

Comme le graphe est non-orienté et les valeurs des arêtes symétriques, un plus cours chemin de i à j est aussi un plus court de j à i lorsqu'il est parcouru à l'envers. En conséquence `distance` est symétrique.

Pour montrer que `distance` satisfait l'inégalité triangulaire, il suffit de noter que, comme le graphe est connexe, il existe un chemin entre i et j (passant par k) de longueur `distance[i][k]` $+$ `distance[k][j]`. Puisque `distance[i][j]` est la longueur du plus petit chemin entre i et j, cette quantité doit nécessairement être au plus la somme précédente.

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

Solution.

On peut utiliser l'algorithme de Floyd-Warshall (programmation dynamique), des produits itérés sur la matrice des distances, ou bien faire des Dijkstra depuis chaque sommet.

Les deux premiers algorithmes utilisent naturellement un tableaux comme `distance[][]`.

La complexité de Floyd-Warshall est $o(n^3)$. Le calcul de la puissance n-ième de la matrice des longueurs d'arêtes peut être faite en $log(n)$ produits. Avec une implémentation naïve du produit de matrices, ça donne $o(n^3.log(n))$; avec une implémentation sophistiquée on peut arriver à $o(n^{2,376}.log(n))$ (ce dernier point est limite hors-programme et n'est pas nécessaire pour avoir la note maximale).

La complexité de Dijkstra dépend du nombre d'arêtes dans le graphe initial. S'il y a N arêtes, l'algorithme de Dijkstra depuis un sommet est en $O((n+N)log(N))$ avec une file de priorité par un tas binaire. On doit itérer cet algorithme n fois.

Si le graphe est relativement dense, $n << N$, il est préférable d'utiliser Floyd-Warshall. Si le graphe est très creux, $N = O(n)$, il est préférable d'utiliser l'itération de Dijkstra pour un coût total en $O(n^2 log(n))$. (Là encore, les détails de cette analyse ne sont pas nécessairement attendus.)

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.*

Solution.

Voici une solution utilisant Floyd-Warshall.

```
  public static void calculDistances() {
    distance = new int[n][n];
```

```
  boolean[][] existeChemin = new boolean[n][n];

  for (int i=0;i<n;i++) {
    LinkedList<Route> l = villes[i].voisins;
    while(! l.isEmpty()) {
      Route r = l.removeFirst();
      int ida=r.a.id, idb=r.b.id, idd=r.distance;
      distance[ida][idb]=idd; distance[idb][ida]=idd;
      existeChemin[ida][idb]=true; existeChemin[idb][ida]=true;
     }
  }

  for (int k=0;k<n;k++)
    for (int i=0;i<n;i++) for (int j=0;j<n;j++) {
      if (existeChemin[i][k] && existeChemin[k][j]) {
        int dk=distance[i,k]+distance[k,j];
        if ((!existeChemin[i][j]) || (dk<distance[i][j])) {
          distance[i][j]=dk; distance[j][i]=dk;
          existeChemin[i][j]=true; existeChemin[j][i]=true;
        }
   }
}
```

PARTIE 2 : Des heuristiques de résolution

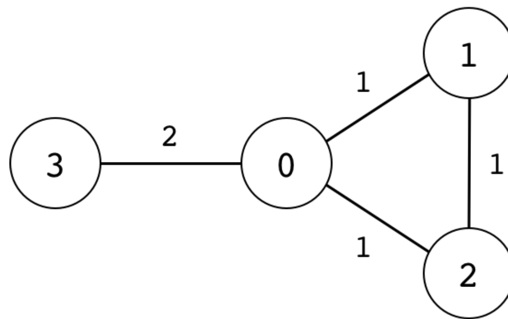
On s'interesse maintenant au graphe complet valué donné par `distance[][]`. On cherche dans ce graphe complet une *tournée*, c'est-à-dire un cycle parcourant une et une seule fois chacune des villes avant de revenir à son point de départ.

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

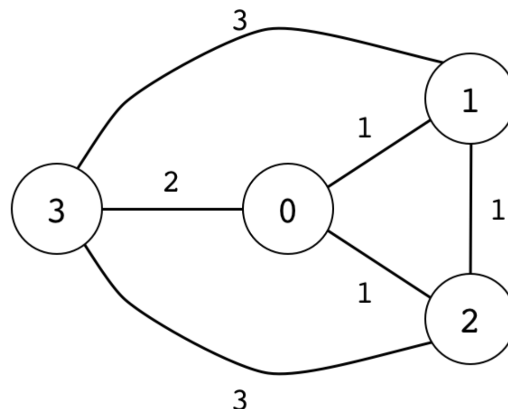
On peut remarquer qu'une arête dans le graphe complet correspond à une ou plusieurs arêtes dans le graphe initial. Aussi, la tournée recherchée dans le graphe complet correspond à un cycle dans le graphe initial, ce dernier pouvant éventuellement passer plusieurs fois par une même ville.

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.

Solution.
Le graphe complet est :



Une tournée optimale dans ce graphe complet est, par exemple : 0, 1, 2, 3, 0.

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 $\{0, 1, \ldots, \texttt{n-1}\}$. Le cycle est donc (`t[0]`, `t[1]`, ..., `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.*

Solution.

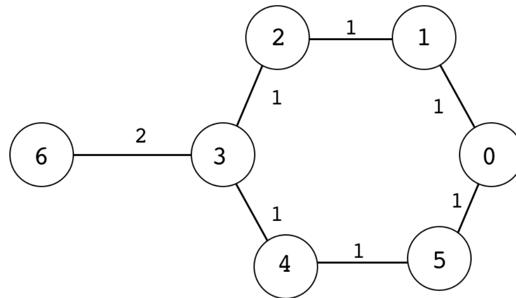
```
public int longueur() {
  int n=t;
  int l=Tsp.distance[t[n-1]][t[0]];
  for (int i=0;i<n-1;i++)
    l=l+Tsp.distance[t[i]][t[i+1]];
  return l;
}
```

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).

Solution.
On peut construire un contre-exemple à partir du graphe donné dans la question 2.1, en rajoutant des sommets dans la "boucle" :



Si on part de 0, on va commencer par parcourir l'hexagone, par exemple 0,1...jusqu'à 5. Il faudra ensuite aller en 6 avant de revenir en 0. Une solution optimale étant : 0,1,2,3,6,4,5,0.

La solution trouvée étant en 14 alors que l'optimale est en 10. Si on augmente le nombre de sommets dans la boucle, on pourrait s'approcher d'une solution deux fois plus mauvaise que la solution optimale.

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 ?*

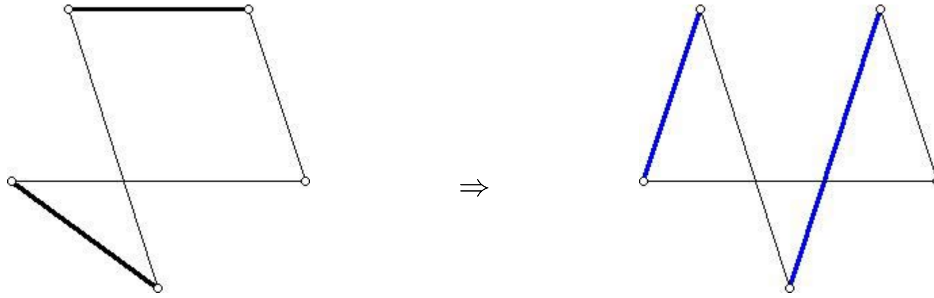
Solution.

```
public void plusProcheVoisin() {
  int n=t.length;
  for (int i=1;i<n-1;i++) {
    int suivant=i;
    for (int j=i+1;j< n;j++)
      if (Tsp.distance[t[i-1]][t[j]]<Tsp.distance[t[i-1]][t[suivant]])
        suivant=j;
    int tmp=t[i];
    t[i]=t[suivant]; t[suivant]=tmp;
  }
}
```

Il y a 2 boucles imbriquées. La complexité est donc en $O(n^2)$.

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* $O(n^2)$*.*

Solution.
La tournée initiale est décrite par le cycle $(t[0], t[1], \ldots, t[n-1])$. Etant donnée une paire (i, j), avec $0 \leq i < j < n$, la tournée modifiée est décrite par le cycle $(t[0], t[1], \ldots, t[i], t[j], t[j-1], \ldots, t[i+1], t[j+1], t[j+2], \ldots, t[n-1])$. Le cas $j = i+1$ correspond au cas de la suppression de deux arêtes adjacentes, et laisse la tournée inchangée. On peut donc supposer que $j \geq i+2$. Posons $jNext = j+1$ si $j < n-1$, et $jNext = 0$ si $j = n-1$. Alors le gain est

$$distance[t[i]][t[i+1]] + distance[t[j]][t[jNext]] - distance[t[i]][t[j]] - distance[t[i+1]][t[jNext]],$$

Pour une tournée donnée, l'algorithme consiste à rechercher une paire (i, j) qui optimise ce gain, puis de renvoyer la tournée modifiée si le gain est positif.

```
public Tournee amelioration() {
   int n=t.length, i=0, gainMax=0, iMax=0,jMax=0;
   for (int i=0;i<n-2;i++) for (int j=i+2;j<n;j++) {
     int jNext=j+1; if (jNext==n) jNext=0;
     int gain=distance[t[i]][t[i+1]]+distance[t[j]][t[jNext]]
                  -distance[t[i]][t[j]]-distance[t[i+1]][t[jNext]];
     if (gain>gainMax) {
       gainMax=gain; iMax=i; jMax=j;
     }
   }
   if (gainMax==0) return null;
   else {
     Tournee nouvelle = new Tournee();
     int i=0;
     for (i=0;i<=iMax;i++) nouvelle.t[i]=t[i];
     i=iMax+1;
     for (int j=jMax;j>=iMax+1;j--) {
       nouvelle.t[i]=t[j];
       i++;
```

```
    }
    for (int j=jMax+1;j<n;j++) nouvelle.t[j]=t[j];
    return nouvelle;
  }
}
```

Il y a 2 boucles imbriquées. La complexité est donc en $O(n^2)$.

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 $\{0, 1, \ldots, n-1\}$ 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 :

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

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

On rappelle que cette notation signifie que les villes sont visitées dans l'ordre $t_1, t_2, \ldots, t_p, t'_1, \ldots, t'_{p'}$, 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 $t'_1 = n-2$.*

Solution.

A cause de la structure pyramidale, la ville $n-2$ est visitée juste avant ou juste après la ville $n-1$. Le fait que `distance` soit symétrique implique que le sens du parcours de la tournée ne change pas sa longueur. Donc quitte à changer le sens de parcours, 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 $j \in \{2, ..., n\}$ un entier donné. La permutation $(t_1, t_2, ..., t_p, t'_1, ..., t'_{p'})$ de $(0, 1, \ldots, j-1)$ est une tournée j-pyramidale si et seulement si $p + p' = j$ et $t_1 < t_2 < \ldots < t_p = j-1$ et $j-2 = t'_1 > t'_2 > \ldots > t'_{p'}$.

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

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

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 $\mathcal{F}_2, \mathcal{F}_3, \ldots, \mathcal{F}_n$?

Solution.

Le terme $\texttt{distance}[t_p][t'_1] = \texttt{distance}[j-1][j-2]$ est manquant dans la définition du coût pour obtenir la longueur de la tournée. La longueur de la meilleure tournée pyramidale est donc $\mathcal{F}_n + \texttt{distance}[n-1][n-2]$.

Question 3.3

On définit pour $2 \leq j \leq n$:

$$\begin{aligned} \mathcal{A}^j &= \texttt{distance}[0][j-1] + \sum_{u=0}^{u=j-3} \texttt{distance}[u][u+1], \\ \mathcal{B}_k^j &= \mathcal{F}_k + \texttt{distance}[k-1][j-1] + \sum_{u=k}^{u=j-3} \texttt{distance}[u][u+1], \quad \textit{pour } 2 \leq k \leq j-1. \end{aligned}$$

Montrez que :

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

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

Solution.

Le cas $j = 2$ est immédiat, car il n'y a qu'une seule tournée 2-pyramidale. Supposons maintenant que $j \geq 3$. Montrons d'abord que $\mathcal{F}_j \leq \min(\mathcal{A}, \mathcal{B}_2, \mathcal{B}_3, \ldots, \mathcal{B}_{j-1})$. La preuve consiste à interpréter chacune des quantités du minimum comme le coût d'une tournée j-pyramidale particulière. Il en résulte donc que $\mathcal{F}_j$ ne peut être que plus petit que chacune de ces quantités.

La quantité $\mathcal{A}$ représente le coût de la tournée $(0, j-1, j-2, \ldots, 1)$. Chaque quantité $\mathcal{B}_k$ représente le coût de la tournée définie à partir d'une tournée k-pyramidale optimale :

1. la tournée commence comme la tournée k-pyramidale jusqu'à la ville $k-1$,
2. puis la tournée visite dans l'ordre les villes $j-1, j-2, \ldots, k-2$,
3. enfin la tournée reprend celle de la tournée k-pyramidale à partir de $k-2$.

Inversement, considérons une tournée j-pyramidale π de coût optimal $\mathcal{F}_j$. Soit $(k-1)$ la ville visitée juste avant n dans la tournée π. Si $k = 1$, alors $\mathcal{F}_j = \mathcal{A}$. Si $k \geq 2$, nous allons montrer que $\mathcal{F}_j = \mathcal{B}_k$. En conclusion, dans tous les cas nous avons $\mathcal{F}_j \geq \min(\mathcal{A}, \mathcal{B}_2, \mathcal{B}_3, \ldots, \mathcal{B}_{j-1})$.

Lorsque $k \geq 2$, soit c le coût de la tournée k-pyramidale extraite de la tournée π ne considérant que les villes d'indice $i < k$. Alors le coût $\mathcal{F}_j$ de π satisfait

$$\mathcal{F}_j = c + \texttt{distance}[k-1][j-1] + \sum_{u=k}^{u=j-3} \texttt{distance}[u][u+1].$$

Par définition, $c \geq \mathcal{F}_k$. Si $c > \mathcal{F}_k$, on pourrait substituer cette portion du tour par une tournée k-pyramidale de coût inférieur. Il en résulterait une tournée j-pyramidale de coût elle-même inférieur à celle de π, ce qui est contradictoire avec l'optimalité de π. Donc $c = \mathcal{F}_k$, et $\mathcal{F}_j = \mathcal{B}_k$.

Question 3.4

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

Solution.

Pour chaque j, le calcul de $\mathcal{A}$ et de chacun des $\mathcal{B}_k$ a une complexité linéaire. Les $\mathcal{B}_k$ étant en nombre linéaire, le calcul d'un nouveau terme $\mathcal{F}_j$, étant donnés $\mathcal{F}_2, \mathcal{F}_3, \ldots, \mathcal{F}_{j-1}$, est donc quadratique.

Puisqu'il faut calculer $\mathcal{F}_k$ jusqu'à $k = n$, la complexité du programme dynamique est en $O(n^3)$.

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 ?

SOLUTION.
Il suffit d'inverser la construction grâce aux interprétations faites de $\mathcal{A}$ et des $\mathcal{B}_k$. Une tournée j-pyramidale de meilleur coût est construite par récurrence à l'aide des meilleurs coûts $\mathcal{F}_2, \mathcal{F}_3, \ldots, \mathcal{F}_j$. Si $\mathcal{F}_j = \mathcal{A}$, la tournée j-pyramidale $(0, j-1, j-2, \ldots, 1)$ convient. Si $\mathcal{F}_j = \mathcal{B}_k$, la tournée j-pyramidale est construite à partir d'une tournée k-pyramidale de meilleur coût à laquelle est greffée la séquence $j-1, j-2, \ldots, k-2$ entre les villes $k-1$ et $k-2$.