UFR IEEA

Algorithmique Avancée et Complexité

2012–2013
Master d'Informatique-S1
TP 3

Résolution pratique du TSP

---

1 Rappel du problème

Dans sa version optimisation, le problème du voyageur de commerce consiste, étant donné un ensemble de n villes séparées par des distances données, à trouver le plus court circuit qui relie toutes les villes.

Voici un exemple de donnée.

	A	B	C	D
A	∞	5	7	8
B	10	∞	16	10
C	7	5	∞	2
D	10	11	17	∞

TABLE 1 – Une matrice de distances possible pour un problème de voyageur de commerce avec quatre villes

2 Heuristiques

On se propose d'écrire deux heuristiques différentes pour le TSP, que l'on comparera expérimentalement.

Une solution pour le TSP est une permutation des villes. On demande de construire deux heuristiques qui vont construire cette permutation de manière différente. Les deux heuristiques sont d'abord décrites, le travail à réaliser est décrit plus bas.

2.1 Construction itérative par ajout du plus proche

La première heuristique construit cette permutation ville par ville en partant du début. Elle peut s'écrire de la manière suivante :

1. Initialiser une permutation avec le premier sommet.
2. A chaque étape choisir la plus proche ville non encore sélectionnée et l'ajouter au tour.
3. S'arrêter lorsque l'on a sélectionné toutes les villes.

Dans cette méthode, on a à toute étape un chemin de longueur inférieure ou égale à n reliant la première ville à une autre. L'algorithme s'arrête lorsque le chemin est de taille $n-1$. Il ne faut pas oublier d'ajouter le retour à la ville de départ.

2.2 Construction par ajout d'arcs

Une autre heuristique peut être utilisée en considérant le TSP comme un problème de graphe. La matrice de distances est alors considérée comme une matrice d'adjacence. On peut alors construire le circuit en ajoutant des arcs itérativement.

1. Trier les arêtes par valeur croissante.
2. A chaque étape, ajouter au tour l'arête courante si elle ne crée par de circuit de taille inférieure à n.

Dans cette méthode, on a à toute étape une liste d'arcs sélectionnés, non obligatoirement consécutifs. Il faut s'assurer que : deux arcs sélectionnés ne partent du même sommet, que deux arcs sélectionnés n'arrivent pas au même sommet, et qu'on ne crée pas de sous-tour (circuit de taille au plus $n-1$).

2.3 Travail à réaliser

Q 1. [A programmer] Implémenter ces deux heuristiques (voir ci-dessous *implémentation*)..
Q 2. Quelles sont les complexités de ces deux méthodes ?
Q 3. Comparer les deux heuristiques sur le jeu de test donné (voir ci-dessous *implémentation*).

Implémentation

Les méthodes à implémenter sont à écrire dans les classes
- `DecreasingEdgeHeuristic.java` et
- `InsertHeuristic.java`.

Seule la méthode `double computeSolution(double[][] matrix, List<Integer> solution);` est imposée par l'interface. Vous pouvez ajouter les méthodes et les constructeurs de votre choix.

Vous pourrez comparer vos méthodes avec la classe `TestTSP` qui permet de lire des fichiers de données, et tester des heuristiques en les lançant et en calculant la moyenne des valeurs obtenues.

Un exemple d'utilisation (que vous pouvez éditer) est contenu dans le fichier `MainTSP.java`.

Les données sont contenues dans le répertoire `data/instances`. Pour le debug, vous pouvez utiliser les instances jouet de `data/toy_instances`. Les données sont entrées comme des coordonnées dans le plan, les classes fournies permettent de transformer ces données en matrices de distance.

3 Borne inférieure

On a montré dans le TP précédent que le problème était difficile dans le cas général, mais on peut parfois déterminer facilement qu'il n'existe pas de solution de valeur inférieure à une certaine valeur.

Calculer une telle valeur a plusieurs intérêts.

1. Dans le cas du problème de décision, il possible de répondre « non » facilement dans certains cas.
2. Cela permet d'estimer la qualité des heuristiques en les comparant à la valeur obtenue.
3. Cela permet d'accélérer des méthodes exactes (voir ci-dessous).

3.1 Description de la borne

Une première borne inférieure peut être obtenue de la manière suivante. Si la distance minimale entre deux villes est k, on ne trouvera jamais de tournée de valeur inférieure à k. On sommant pour toutes les villes, on obtient une borne égale à nk.

On peut généraliser le raisonnement de la manière suivante. On sait que dans un tour, on va devoir emprunter un chemin sortant de chacune des villes. Par conséquent la somme des distances minimales sortant de chaque ville est une borne inférieure pour la valeur optimale d'un tour.

La méthode suivante est basée sur cette idée.

1. pour chaque ligne de la matrice de distances, soustraire la valeur minimum sur cette ligne
2. pour chaque colonne de la matrice obtenue, soustraire la valeur minimum sur cette colonne

	A	B	C	D
A	∞	5	7	8
B	10	∞	16	10
C	7	5	∞	2
D	10	11	17	∞

	A	B	C	D
A	∞	0	2	3
B	0	∞	6	0
C	5	3	∞	0
D	0	1	7	∞

	A	B	C	D
A	∞	0	0	3
B	0	∞	4	0
C	5	3	∞	0
D	0	1	5	∞

TABLE 2 – Prétraitement sur la matrice de distances : la matrice initiale, la matrice obtenue en soustrayant le minimum sur chaque ligne et la matrice obtenue en soustrayant le minimum sur chaque colonne. La borne est égale à $(5 + 10 + 2 + 10) + 2 = 29$.

On remarque que cette méthode permet à la fois d'obtenir une borne inférieure et de simplifier la donnée (on remarque qu'on obtient au moins un zéro par ligne et par colonne).

3.2 Travail à réaliser

Q 4. Montrer que toute solution optimale du problème réduit est solution optimale du problème initial.
Q 5. Comment obtenir la valeur de la solution du problème initial en fonction de la valeur de la solution du problème réduit ?
Q 6. [A programmer] Ecrire une méthode qui calcule une borne inférieure pour le problème de tournée et qui calcule une donnée simplifiée.
Q 7. Quelle est la complexité de cette méthode ?

Implémentation

Vous pouvez éditer le fichier `LowerBound.java` du paquetage `tsp`.

4 Méthode exacte

On se propose d'écrire une méthode de résolution exacte basée sur une recherche arborescente. Il s'agit de l'algorithme de Little. Pour éviter une énumération complète, on ajoute une méthode qui va évaluer pour chaque nœud de la recherche s'il peut mener à une meilleure solution. Le travail à effectuer est indiqué pour chaque étape.

4.1 Version basique de la recherche arborescente

À un nœud donné de l'arbre de recherche, on dispose d'une liste d'arcs sélectionnés et d'une matrice courante de distances.

Un nœud correspond à une solution partielle (une liste d'arcs sélectionnés). A la racine, la liste est vide. Au cours de l'énumération, on va ajouter ou interdire des arcs jusqu'à obtention d'une solution ou épuisement des arcs.

Le schéma de séparation est le suivant : à partir du nœud courant, on sélectionne un arc (i, j) (ni ajouté ni interdit) de valeur différente de $+\infty$ et on crée deux nouveaux nœuds (qu'on appellera "fils" du nœud courant) : l'un où l'on ajoute (i, j), l'autre où l'on interdit d'ajouter (i, j).

Dans un premier temps, on peut choisir le premier couple (i, j) tel que $M_{ij} \neq +\infty$.

Q 8. Si on sélectionne l'arc (i, j), quels arcs peut-on interdire automatiquement ? On pensera notamment à l'arc (j, i).
Q 9. Comment évolue le coût minimum de la solution courante lorsque l'on ajoute un arc ?
Q 10. Comment modifier le problème dans le deuxième fils pour s'assurer que l'arc (i, j) ne sera pas choisi ?

On est arrivé à une feuille lorsqu'on a obtenu n arcs dans la solution partielle. Une feuille correspond à une solution réalisable s'il n'y a pas de sous-tour.
Q 11.[A programmer] Implémenter la recherche arborescente basique (voir ci-dessous **implémentation**).

4.2 Introduction de la borne inférieure

On peut améliorer la méthode en stoppant l'évaluation d'un nœud de l'arbre lorsque la valeur de sa solution partielle est supérieure à la meilleure solution déjà trouvée.
Q 12. Lorsqu'un arc est interdit, comment mettre à jour la borne de la section 3 ?
Q 13.Même question lorsqu'un arc est ajouté. Ne pas oublier que l'ajout d'un arc provoque l'interdiction d'autres arcs.
Q 14.[A programmer] Intégrer cette borne dans votre algorithme.

4.3 Choix de la séparation

Le choix du couple (i, j) sur lequel la séparation s'effectue est important pour l'efficacité de la méthode.

Si l'on veut arriver rapidement à une bonne solution, une première idée est de choisir toujours un arc dont la valeur est zéro (il en existe obligatoirement si on a utilisé la technique de borne inférieure décrite plus haut).

Parmi les arcs de valeur 0, on choisira celui qui occasionnerait le plus grand surcoût s'il n'était pas choisi. Pour ce faire, pour chaque arc (i, j) de coût 0, on calcule la valeur minimum d'un arc $(v, j), v \neq i$ et $(i, v), v \neq j$.
Q 15.[A programmer] Introduire le choix du meilleur arc à insérer dans la méthode.

Implémentation

Vous pouvez utiliser la classe `BranchAndBound` du paquetage `branchAndbound`. Pour utiliser cette classe, vous devrez éditer le fichier `NodeTSP.java` du paquetage `tsp`.

Vous pouvez tester vos algorithme sur les données incluses dans `data/toy_instances`.