INFO-F310 - Algorithmique et recherche opérationnelle

Dimitrios Papadimitriou
dimitrios.papadimitriou@ulb.be

Hugo Callebaut
hugo.callebaut@ulb.be

1 Problème

Le problème du voyageur de commerce (Traveling Salesman Problem ou TSP) est l'un des problèmes d'optimisation combinatoire les plus étudiés en recherche opérationnelle. Un voyageur doit visiter un ensemble de n villes exactement une fois chacune et revenir à son point de départ, en minimisant la distance totale parcourue.

Formellement, étant donné un ensemble de n villes $V = \{0, 1, \ldots, n-1\}$ et une matrice dont les éléments $c_{ij} \geq 0$ représentent le coût (distance, temps, etc.) pour aller de la ville i à la ville j, l'objectif est de trouver un cycle hamiltonien de coût minimum.

Le TSP possède de nombreuses applications pratiques : optimisation de tournées de livraison, perçage de circuits imprimés, séquençage ADN, planification de trajets de drones, et bien d'autres domaines de la logistique et de l'industrie.

2 Formulations du problème

Il existe plusieurs formulations en programmation linéaire en nombres entiers pour le TSP. On vous demande d'étudier deux formulations classiques :

— La formulation de Miller-Tucker-Zemlin (MTZ) [1]. Cette formulation utilise des variables auxiliaires pour éviter les sous-tours.

— La formulation de Dantzig-Fulkerson-Johnson (DFJ) [2]. Cette formulation utilise des contraintes d'élimination de sous-tours pour chaque sous-ensemble de villes.

Définition : Un *sous-tour* est un cycle qui visite strictement moins de n villes. Une solution du TSP est valide si et seulement si elle forme un unique cycle visitant toutes les n villes (cycle hamiltonien).

Note : Vous êtes encouragés à consulter des ressources pour bien comprendre ces deux formulations avant de les implémenter. Par exemple, ces formulations sont documentées sur la page Wikipedia du TSP `https://en.wikipedia.org/wiki/Travelling_salesman_problem#Integer_linear_programming_formulations`

3 Génération itérative de contraintes pour DFJ

Comme la formulation DFJ nécessite un nombre exponentiel de contraintes, il est peu pratique de les générer toutes a priori pour des instances de taille raisonnable. Une approche classique consiste à utiliser la génération itérative de contraintes.

1. C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer Programming Formulation of Traveling Salesman Problems. J. ACM, 7(4) :326–329, Oct. 1960

2. G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. Journal of the Operations Research Society of America. 2(4) :393-410, Nov. 1954

3.1 Principe de l'algorithme

L'algorithme fonctionne comme suit :

Algorithm 1 Génération itérative de contraintes pour DFJ

Créer le modèle PuLP (DFJ) sans les contraintes d'élimination de sous-tour
$iteration \leftarrow 0$
while true **do**
    Résoudre le modèle actuel
    Extraire la solution (ensemble des arcs (i, j) tels que $x_{ij} = 1$)
    Détecter tous les cycles dans la solution
    if il existe un unique cycle visitant toutes les n villes **then**
        break - Solution optimale trouvée
    else
        for chaque cycle (i.e. sous-tour) $S, |S| \geq 2$ détecté **do**
            Ajouter la contrainte : $\sum_{i \in S} \sum_{j \neq i : j \in S} x_{ij} \leq |S| - 1$
        end for
        $iteration \leftarrow iteration + 1$
    end if
end while

Pour résumer, vous commencez par résoudre le problème sans aucune contrainte d'élimination de sous-tour. Ensuite vous vérifiez si votre solution contient plusieurs cycles, si non, elle est optimale, si oui, vous ajoutez pour chaque cycle trouvé une contrainte d'élimination de sous-tour correspondant au cycle et recommencez à la résolution du problème.

À vous d'implémenter une fonction permettant de détecter tous les cycles présents dans une solution donnée.

4 Instances

Format des instances

Un répertoire contenant des instances `instance_n_type_id.txt` du problème vous est fourni. Les fichiers sont au format suivant :

- `n` correspond au nombre de villes dans l'instance
- `type` indique le type d'instance (`euclidean`, `line`, `circle`, `random_sym`, `random_asym`)
- `id` est un identifiant unique pour distinguer plusieurs instances de même taille et type

Chaque fichier contient :

- La première ligne : un entier n représentant le nombre de villes
- Les n lignes suivantes : pour chaque ville i, ses coordonnées x_i et y_i séparées par un espace
- Les n lignes suivantes : la matrice de distances c_{ij}, avec n valeurs par ligne séparées par des espaces

Types des instances

- `euclidean` : Villes placées sur une grille 2D aléatoirement, distances euclidiennes
- `circle` : Villes disposées en cercle, distances euclidiennes
- `line` : Villes alignées, distances euclidiennes
- `random_sym` : Distances générées aléatoirement et symétriques
- `random_asym` : Distances générées aléatoirement et asymétriques

5 Tâches

On vous demande de réaliser :

1. **Implémentation des formulations en PuLP**
   - Implémenter la formulation MTZ
   - Implémenter la formulation DFJ en énumérant toutes les contraintes d'élimination de sous-tours (DFJ énumératif)
   - Implémenter la formulation DFJ avec génération itérative de contraintes d'élimination de sous-tours (DFJ itératif)
   - Pour chaque formulation (MTZ et DFJ), implémenter le programme linéaire en nombre entier et pour MTZ et DFJ énumératif (pas DFJ itératif) sa relaxation continue. Cette relaxation s'obtient en remplaçant les contraintes d'intégralité sur les variables du programme linéaire en nombres entiers par des contraintes continues : $x_{ij} \in \{0,1\} \rightarrow 0 \leq x_{ij} \leq 1,\ \forall i,j \in \{1,\ldots,n\}$. Toutes les variables étant à présent continues, cette relaxation est donc un programme linéaire.
2. **Compréhension des mécanismes d'élimination de sous-tours**

   Pour chacune des deux formulations (MTZ et DFJ) :
   - Expliquer en vos propres mots comment les contraintes spécifiques empêchent la formation de sous-tours, puis illustrer avec un exemple concret.
   - Construire un exemple avec 5 villes où une solution avec 2 sous-tours violerait les contraintes spécifiques mais pas les contraintes communes. Votre exemple doit inclure :
     - Un schéma montrant les 5 villes et les arcs formant deux sous-tours
     - Les valeurs hypothétiques des variables u_i pour chaque ville ou l'identification des sous-ensembles S concernés
     - Une explication montrant qu'au moins une contrainte spécifique est violée

     Note : Les contraintes communes sont les contraintes de degré (pour chaque ville exactement un arc entrant et un arc sortant doivent être sélectionné). Les contraintes spécifiques sont celles qui diffèrent entre MTZ et DFJ pour empêcher les sous-tours.
3. **Comparaison des formulations MTZ et DFJ**
   - Pour chaque formulation (MTZ et DFJ), créer un tableau comparatif montrant pour chaque instance (pour DFJ énumératif, limitez-vous au instances de maximum 15 sommets, les autres seront trop longues à résoudre) :
     - Le nombre de variables
     - Le nombre de contraintes (pour DFJ itératif : nombre final après génération itérative)
     - Le temps de résolution du programme linéaire en nombres entiers
     - La valeur objective (vérifier que les valeurs obtenues pour les formulations MTZ et DFJ sont identiques)
   - Analyser ces résultats : identifier des patterns, expliquer les différences observées, et tirer des conclusions sur les forces et faiblesses de chaque formulation.
4. **Analyse de la relaxation continue**
   - Pour chaque formulation (MTZ et DFJ) et pour chaque instance (à nouveau pour DFJ énumératif, limitez-vous au instances de maximum 15 sommets)
     - Résoudre la relaxation continue du programme linéaire en nombres entiers (cf. Tâche 1)
     - Comparer la valeur de la solution de la relaxation continue avec celle de la solution entière optimale en calculant le *integrality gap* relatif : $\frac{\text{valeur de la solution entière} - \text{valeur de la relaxation}}{\text{valeur de la solution entière}}$
   - Analyser ces résultats : identifier des patterns, expliquer les différences observées, et tirer des conclusions sur les forces et faiblesses de chaque formulation.
5. **Questions de réflexion**
   - On pourrait en théorie résoudre le TSP en énumérant tous les $(n-1)!/2$ cycles possibles. La formulation DFJ, elle, considère 2^n sous-ensembles. Les deux approches sont exponentielles, pourtant l'une est praticable et l'autre non. Analyser et expliquer cette différence fondamentale.

— La formulation DFJ nécessite un nombre exponentiel de contraintes ($O(2^n)$), tandis que MTZ n'en a qu'un nombre quadratique ($O(n^2)$). Pourtant, DFJ peut être plus rapide en pratique. Expliquer ce comportement en vous basant sur vos observations et analyses.

6. **Bonus** (optionnel pour 2 points sur 20)
   — Lorsqu'il y a exactement 2 sous-tours détectés, montrer mathématiquement qu'ajouter une seule contrainte de sous-tour suffit (l'autre contrainte est redondante). Vérifier expérimentalement en modifiant votre code DFJ pour n'ajouter qu'une seule contrainte dans ce cas. Le nombre d'itérations et le temps de résolution changent-t-ils ?

NOTE IMPORTANTE : En ce qui concerne la mesure de temps, pour une comparaison équitable, mesurez uniquement le temps passé dans le solveur (appels à `prob.solve()`), en excluant la génération de contraintes en Python [3]. Pour DFJ itérative, additionnez les temps de tous les appels au solveur.

6 Documents à remettre

1. **Scripts Python** :
   Un script python3 nommé `tsp_solver.py` prenant en paramètres en ligne de commande :
   — Le nom d'un fichier d'instance
   — Un paramètre f indiquant la formulation à utiliser :
      — $f = 0$: Résoudre MTZ
      — $f = 1$: Résoudre la relaxation continue de MTZ
      — $f = 2$: Résoudre DFJ avec toutes les contraintes d'élimination de sous-tours générées à priori
      — $f = 3$: Résoudre la relaxation continue de DFJ avec toutes les contraintes d'élimination de sous-tours générées à priori
      — $f = 4$: Résoudre DFJ avec la génération itérative de contraintes

   Le script doit afficher :
   — La valeur de la fonction objectif
   — Le cycle hamiltonien trouvé (pour les solutions entières)
   — Le temps de résolution (en secondes)
   — Pour DFJ itérative : le nombre d'itérations

   Exemple d'appel :

   ```
   python3 tsp_solver.py instance_10_random_sym_1.txt 4
   ```

   Le script doit utiliser la librairie PuLP pour la modélisation et la résolution.

2. **Rapport scientifique** :
   Un rapport LaTeX compilé au format `pdf` qui détaille vos formulations, implémentations et analyses. Le rapport doit contenir au minimum :
   — Vos noms, prénoms, matricules et année d'étude
   — Des réponses complètes et détaillées à toutes les questions de la Section 5
   — Des tableaux, graphiques et figures pour présenter vos résultats de manière claire
   — Une analyse critique de vos résultats avec un esprit scientifique
   — Toutes les hypothèses que vous avez faites

   Important : Nous insistons sur le fait de remettre un **rapport scientifique**. Une analyse approfondie et critique de vos résultats est attendue. Les phrases génériques sans lien avec vos résultats spécifiques ne seront pas valorisées.

3. Le Python est interprété et environ 10-100× plus lent que les solveurs optimisés (compilés en C/C++). Inclure le temps de génération de contraintes en Python fausserait la comparaison : DFJ énumérative souffrirait de la génération de milliers de contraintes, et DFJ itérative des appels répétés au solveur et de l'implémentation de la détection de cycles. En mesurant uniquement le temps solveur, on compare l'efficacité des formulations elles-mêmes, pas les détails d'implémentation.

3. **Résultats** :
   Un fichier `results.csv` contenant vos résultats obtenus sur chaque instance demandée avec chaque formulation (à nouveau ne résolvez pas les instances de 20 sommets ou plus avec DFJ énumératif). Ces fichiers doivent contenir au minimum les colonnes suivantes :
   - Nom de l'instance
   - Formulation utilisée (MTZ, DFJ_enum, DFJ_iter)
   - Valeur objective de la solution entière
   - Temps de résolution du programme linéaire en nombres entiers (en secondes)
   - Valeur objective de la relaxation continue (si applicable)
   - Temps de résolution de la relaxation continue (en secondes, si applicable)
   - Integrality gap (pour MTZ et DFJ_enum)
   - Nombre de variables
   - Nombre de contraintes (pour DFJ itérative : nombre final)

   Vous pouvez ajouter toute autre information que vous jugez pertinente pour vos analyses.

   Note : Les valeurs dans ces fichiers doivent correspondre exactement aux résultats présentés dans votre rapport.

   Exemple de format attendu :

```
instance,formulation,obj_int,time_int,obj_relax,time_relax,gap,vars,constr
instance_10_euclidean_1,MTZ,245.3,0.12,240.1,0.03,0.021,100,190
instance_10_euclidean_1,DFJ_enum,245.3,0.18,243.5,0.04,0.007,100,1012
instance_10_euclidean_1,DFJ_iter,245.3,0.08,243.5,0.02,0.007,100,45
```

7 Consignes de remise

Ce travail peut être réalisé individuellement (1) ou par groupe de deux (2) étudiants (recommandé).
Pour soumettre votre projet sur l'UV du cours INFO-F310, nous vous demandons de :

- Créer sur votre machine un répertoire local intitulé `NOM1_NOM2` (exemple : `PAPADIMITRIOU_CALLEBAUT`, sans espace) dans lequel vous placez les fichiers demandés (scripts, rapport PDF, results.csv, sans y inclure les fichiers d'instances fournis)
- Compresser ce répertoire via un utilitaire d'archivage produisant un fichier `.zip` (aucun autre format de compression n'est accepté)
- Soumettre le fichier archive `.zip`, et uniquement ce fichier, sur l'UV (une seule soumission par groupe)

Le projet est à remettre pour le **[22-12-2026]** sur l'UV.
Tout manquement aux consignes ou retard sera sanctionné directement d'une note de **0/20** pour le projet.
Il est à souligner que toute tentative de fraude ou de plagiat détectée dans le code ou le rapport sera sévèrement sanctionnée (dans ce dernier cas, pour tous les groupes impliqués).

Le code Python compte pour 20% de la note du projet et le rapport scientifique pour 80% de cette note. Veuillez tenir compte que la note du projet est automatiquement reportée pour la note finale de seconde session.

8 Conseils

- Utilisez des structures de données efficaces (dictionnaires Python) pour représenter les graphes
- Pour la visualisation, vous pouvez utiliser `matplotlib` (optionnel mais recommandé)
- Dans votre rapport, privilégiez l'analyse de **vos résultats** plutôt que des généralités théoriques