GOSSET Guillaume, LASOTA Quentin
PICHARD Thomas, SCION Gaël

Projet d'Optimisation Stochastique

Problème du voyageur de commerce

Document technique

Enseignant encadrant :

Mme KHEBBACHE Selma

Sommaire

I. Présentation du problème du voyageur de commerce

A. Description des données intéressantes pour la modélisation

Le problème du voyageur de commerce est un problème de théorie des graphes visant à déterminer dans un ensemble de points donné, le plus court chemin qui visite chaque point une et une seule fois et qui se termine en revenant dans la ville de départ. Il s'agit d'un problème d'optimisation NP-complet du fait du grand nombre de combinaisons et d'arrangements possibles (de l'ordre de n!) qu'il est nécessaire d'explorer pour trouver la solution optimale.

On représente chacune des villes à parcourir comme les sommets d'un graphe. Chaque sommet stockant sa position géographique sous forme de coordonnées dans un espace en 2D.

Le graphe considéré est un graphe complet orienté, c'est-à-dire un graphe dont toutes les paires de sommets sont reliées deux à deux entre elles. De plus, le graphe étant orienté, les arcs reliant deux sommets entre eux peuvent donc avoir un coût différent.

On considère que l'on peut rejoindre n'importe quelle ville depuis n'importe quelle autre ville sans passer par une ville intermédiaire, peu importe sa position dans l'espace.

II. Présentation du modèle mathématique

A. Variables du modèle

- c : matrice de distance entre i et j
- x : matrice permettant d'indiquer si l'arc est emprunté dans le circuit (elle ne prend qu'une valeur de 0 ou de 1).

B. Fonction objectif et contraintes du modèle

- Fonction objectif : Trouver le plus court chemin en moyenne.
  On calcul le coût global en faisant la somme des distances moyennes des arcs qu'on a sélectionné. Les arcs non sélectionnés sont ignorés (car leur xij vaut 0 dans la matrice)
- Contrainte (1a) : Permet de vérifier que tout sommet a exactement une sortie (présence d'un unique élément suivant)
- Contrainte (1b) : Permet de vérifier que tout sommet a exactement une entrée (présence d'un unique élément précédent)
- Contraintes (1a) et (1b) permettent de vérifier qu'on considère un circuit hamiltonien, c'est-à-dire avec une entrée et une sortie pour chaque sommet.
- Contrainte (1c) : Pour tout sous-graphe S, le nombre d'arcs est inférieur au nombre de sommets - 1 dans ce sous-graphe. Le sous-graphe est un ensemble de sommets. Cela permet de s'assurer que l'on explore la totalité des voisinages possibles, et permet de s'assurer de l'exactitude de la solution.
- Contrainte (1d) : Cette contrainte représente la limite de probabilité minimale (alpha) que l'on souhaite avoir pour s'assurer que la solution probabiliste qu'on obtient soit au moins aussi bonne (aussi faible) que la solution déterministe majorée de 20 à 30%.
- Contrainte (1e) : Représente les contraintes d'intégrité, où les éléments de x sont des booléens, et i,j sont bien compris entre 1 et n pour correspondre à des indices des matrices c et x.
-

III. Description du modèle de recuit simulé.

Dans le cadre de notre modèle de recuit simulé, nous avons défini comme voisinage de la solution courante les différentes solutions qu'on obtiendrait avec une ou plusieurs permutations de sommets à l'intérieur. Dans le cadre de notre algorithme, nous avons défini plusieurs moyens de parcourir ce voisinage : soit faire un échange de deux valeurs aléatoires dans la solution, soit de faire des échanges successifs de paire de valeurs en divisant par 2 la probabilité de continuer à faire des échanges à chaque fois, ou soit de faire des échanges de deux groupes de 3 valeurs côte à côte. Pour générer la solution initiale utilisée dans l'algorithme nous avons choisi d'implémenter 2 solutions, une génération purement aléatoire, et une solution générée à partir d'une heuristique gloutonne.
Le premier réglage nécessaire pour l'algorithme du recuit simulé concerne la température de départ. Parmi nos différents essais, une température de départ aux alentours de 200 à 500 degrés permettent d'obtenir des bons résultats pour pouvoir parcourir une bonne partie des zones convexes de l'ensemble des solutions sans dégrader de manière trop importante les temps de calcul.

Le deuxième paramètre que nous avons à régler dans le recuit simulé est la température d'arrêt. Elle doit être suffisamment grande pour ne pas rallonger de manière trop importante le temps de calcul, mais suffisamment petite pour permettre une phase d'intensification assez importante vers la fin de l'exécution de l'algorithme. Au cours des différents tests que nous avons fait, nous avons retenu une valeur finale entre 0.5 pour les cas les plus long et 0.05 pour les cas les plus rapides, afin de permettre une meilleure phase d'intensification.

Le troisième paramètre concerne la décroissance de la température. Nous avons décidé de l'appliquer à l'aide d'un facteur multiplicateur que l'on multiplie à la température à chaque étape du calcul. Encore une fois, ce facteur doit permettre une réduction progressive de la température suffisamment lente pour que l'on passe suffisamment de temps à haute température pour avoir une bonne exploration de l'espace, et suffisamment basse pour ne pas passer un temps trop important dans la phase d'intensification. Nous avons retenu une valeur de 0.99999.

Concernant la partie stochastique de notre programme, nous avons pour l'instant fait le choix d'incorporer la variation de la distance de la solution au moment de son calcul (durant l'exécution du recuit simulé). De cette manière, la distance utilisée pour comparer la nouvelle solution trouvée et la solution de référence incorporent cette part d'aléatoire. Cependant, cette représentation n'est pas idéale car le calcul de la distance de la solution n'est pas constante lorsqu'on l'effectue deux fois de suite, c'est donc pour cela que nous avons choisi de retravailler cet aspect pour obtenir un fonctionnement qui est plus

consistant et qui se rapproche plus du fonctionnement de la contrainte stochastique telle qu'elle est décrite dans le modèle pour qu'elle soit prise en compte d'une façon plus adaptée. Cette mise en place a nécessité le calcul d'une répartition de la valeur possible de la distance selon la loi normale. Pour ce faire, nous avons dû définir la dernière variable de paramétrage du projet : le facteur permettant de définir l'écart-type de cette loi normale. Nous avons en effet fait le choix d'exprimer l'écart type de la variable comme une fraction de sa moyenne (de telle sorte à ce que des valeurs plus grandes soient plus variables que des petites, mais de manière proportionnelle). Dans le cas de notre recuit simulé, nous avons fixé cette valeur à 0.025 de telle sorte à ce que l'écart type de la loi normale soit d'environ 2.5% de la moyenne, car en testant, nous avons remarqué que cela permettait d'avoir une bonne variabilité en prévenant les valeurs extrêmes pouvant totalement fausser le calcul.
Dans un deuxième temps nous avons fait le choix d'une implémentation qui se rapproche plus de la contrainte originelle comme elle a été définie dans le modèle mathématique. En effet, avant de valider une meilleure solution (en plus de comparer sa moyenne avec celle de la référence courante), on effectue 100 tirages d'une nouvelle distance selon une répartition de la loi normale adaptée à ce chemin. Par la suite, on compare la proportion de ces 100 tirages qui sont inférieure à la moyenne de la référence actuelle majorée de 30%. Si cette proportion est supérieure à 90% des tirages, on accepte la solution, sinon on la rejette. Dans tous les cas, si le tirage probabiliste permettant l'acceptation d'une solution plus mauvaise, permet l'acceptation de la solution, elle sera acceptée sans réaliser cette vérification stochastique, afin de diminuer la complexité de l'algorithme.

IV. Description de la résolution sous CPLEX

Pour ce qui est des données, on les re-formate. En partant de la liste des N points et de leurs coordonnées, on crée une matrice N*N qui contient les distances (coûts) entre chaque points (pour chaque couple <i,j>). On obtient un fichier .dat, lisible par OPL, précisant le nombre de points et cette matrice.
Pour chacun de ces graphes, on associe également un fichier se terminant par "_subtour.dat" qui contient la totalité des sous-ensembles possibles (sauf le graphe complet et l'ensemble vide).
Les deux fichiers .dat sont générés en partant de la structure des données de l'annexe du projet et en exécutant le programme subgraph_gen.py.

A. Partie déterministe

Pour la partie déterministe, on détermine tout d'abord les données et variables :

- nbVertex, nombre de sommets, ainsi que Vertex, un intervalle allant de 1 à nbVertex.
- c, tableau d'entiers contenant les coûts (distances), de taille nbVertex x nbVertex.
- x, variable à déterminer, qui représente une matrice de booléens de taille nbVertex x nbVertex.
- Subtours, constitué de la liste des sous-ensembles du graphe (de cardinal supérieur à 1 et strictement inférieur à nbVertex).

Pour la fonction objectif, il s'agit de la somme en x et c, équivalent à la somme des coûts des chemins qui seront sélectionnés. On souhaite minimiser cette fonction (car on minimise les coûts, c'est-à-dire la distance).

Pour ce qui est des contraintes, on retrouve :

- l'ignorance des boucles, en précisant x[v,v] égal à 0 pour tout v.
- l'unicité de l'entrée et de la sortie pour chaque sommet, en précisant que pour tout i (respectivement j), la somme sur j (respectivement sur i) des x[i,j] est égale à 1. Avec la contrainte précédente, cela correspond au (1a) et (1b) du modèle mathématique de l'énoncé.
- pour la contrainte des sous-ensembles (1c), on parcourt la totalité des sous-ensembles (générés dans un fichier extérieur), et on vérifie que le nombre d'arcs assignés d'un sous-ensemble est plus faible que le nombre de sommets du sous-ensemble. Cela permet de s'assurer que le chemin est bien continu.
- l'intégrité des valeurs de x(i,j), qui doivent valoir soit 0 soit 1 pour chaque couple

Par ailleurs, la contrainte d'intégrité des indices est assurée par le forall où on indique que l'itération se fait sur la range Vertex = {1, ..., nbVertex}.

B. Partie stochastique

Pour la partie stochastique, on récupère les mêmes variables, et on rajoute les suivantes :

- V, matrice variance-covariance, de taille nbVertex x nbVertex.
- alpha, pourcentage de cas où le risque est satisfait.
- stdDevCoef, coefficient de la moyenne auquel on associe l'écart-type (par exemple, si la valeur est de 0.1, l'écart-type sera de 0.1 * moyenne).
- opti, optimum généré lors de la solution déterministe, et fournie en paramètre (grâce au main.mod)
- Z, majoration de 30% de la variable opti.

On génère également la matrice de variance-covariance en pré-processing de CPlex en déclarant un écart-type comme étant un pourcentage de la moyenne (ce pourcentage est géré par stdDevCoef, en source de main.mod), et en prenant le carré de cet écart-type pour chaque arc <i,j>. La valeur de alpha est également récupérée en source dans le fichier main.mod.

On retrouve la même fonction objectif et les mêmes contraintes, auxquelles on ajoute la suivante :

- contrainte stochastique, pour laquelle la probabilité d'obtenir une distance inférieure à Z (optimum majoré de 30%) est égale à alpha. Pour se faire, on utilise l'approximation de la fonction de répartition inverse (*fonction quantile*) de **Shore**, qui nous donne la valeur Z pour l'entrée alpha qui répond à P(N(0.1,) <= Z) = alpha.

Pour pouvoir utiliser cette fonction, il faut d'abord centrer et réduire notre loi, ce qui revient à centrer et réduire Z :

$$\mathbb{P}(Y \leq x) = \mathbb{P}\left(X \leq \frac{x-\mu}{\sigma}\right)$$

avec μ la moyenne des coûts (c barre), et σ l'écart type des coûts (racine carrée de la variance)

On obtient donc :

$$Z - c.x <= 5.5556 * (1 - ((1-\alpha)/\alpha)^{0.1185}) * \sqrt{V}.c$$

Nous avons cependant préféré calculer la fonction quantile sur alpha en pré-processing, et la stocker dans la variable quantileAlpha :

$$Z - c.x <= quantileAlpha * \sqrt{V}.c$$

C. Formulation de Miller–Tucker–Zemlin

Cette formulation permet un modèle différent pour le problème de voyageur de commerce. On remplace la contrainte (1c) des sous-ensembles par la suivante :

$$u_i - u_j + nx_{ij} \leq n - 1 \qquad 2 \leq i \neq j \leq n;$$
$$1 \leq u_i \leq n - 1 \qquad 2 \leq i \leq n.$$

On associe un “rang” à chaque ville, et la contrainte ordonne les rangs selon le chemin, afin d’éviter d’avoir plusieurs chemins (le but est le même que pour la contrainte des sous-ensembles, mais ce modèle est plus rapide et moins coûteux en mémoire).

Pour cela, on ajoute la variable u, tableau allant de 2 à nbVertex, représentant ce “rang”.
On ajoute également la contrainte pour laquelle, si l’arc (i,j) est sélectionné, alors le rang de i est inférieur au rang de j. On obtient donc bien que le rang est lié à l’ordre des villes pour le chemin.

Par ailleurs, on ne représente pas le rang de la première ville étant donné que le chemin boucle, donc le premier ou le dernier arc ne satisferait pas la contrainte (on aurait u(1) < u(i) et u(j) < u(1) mais tout en ayant u(i) < u(j) par propagation de la contrainte dans le chemin). En effet, le but de la condition est d’empêcher les boucles sauf pour la première ville.