Métaheuristique pour l'optimisation

Série 2: The Quadratic assignment problem

Romain Mencattini

11 octobre 2016

Table des matières

1 Quadratic Assignment Problem

Nous allons expliquer le problème afin de bien saisir les enjeux du problème. Les problèmes d'assignation quadratiques sont une famille de problèmes qui demandent d'assigner des objets ou *facilities* à une place ou *locations*. On peut imaginer un exemple, (repris dans l'énoncé du tp) où les objets sont des usines et les places des villes. On souhaite placer au mieux les usines dans les dites villes.

Le critère de qualité est une fonction F qui est définie comme suit :

$$arg \min_{\psi \in S(n)} I(\psi) = \sum_{i,j=1,\dots,n} w_{ij} \cdot d_{\psi_i,\psi_j}$$

Où w_{ij} correspond au flux entre l'objet i et l'objet j, et d_{ψ_i,ψ_j} correspond à la distance entre la place de l'objet i et la place de l'objet j.

Ces éléments sont donnés par deux matrices. La matrice W et la matrice D qui sont de tailles n x n. Avec n qui est la taille de notre solution. Les matrices W, D sont symétriques et leur diagonale est nulle.

Le but est de trouver un vecteur de taille n qui à chaque place fait correspondre un objet de manière que la fonction F soit minimale.

2 Tabu Search

Expliquons maintenant la recherche tabou ainsi que ses subtilités. La recherche se déroule comme suit :

2.1 Génération d'une solution

On génère une solution aléatoire. La solution sera de taille n. Avec les n objets placés de manière aléatoire sur les n places du vecteur solution. On pose que $best = F(solution)$. Car on désire garder en mémoire le meilleur résultat rencontré.

2.2 Génération de voisins

Pour une solution donnée, soit x_0. On veut calculer $V(x_0)$. Pour ce faire, on définit que les voisins de x_0, qui sont tous les solutions égalent à x_0 à une permutation près. Donc :

$$x_0 = (1, 2, 3, 4)$$
$$x_1 = (2, 1, 3, 4) \in V(x_0)$$

Une fois ces voisins générés, on va les trier selon leur fitness. On applique donc $F(x_i)$, où $x_i \in V(x_0)$. Et on trie selon ce critère.

2.3 Choix du voisins

Le choix du voisins se fait de la manière suivante :

1. Si $F(x_i) < F(\text{best})$, alors on choisit x_i, où x_i est notre meilleur voisin

2. Si la condition 1. n'a pas fonctionné, on prend le meilleur voisin non tabou.

Une fois ce choix fait, on met à jour *best*.

2.4 Liste Tabou

Dans le cas de ce programme, on a implémenté la liste tabou comme étant une matrice de taille n x n. Après l'échange de l'objet i en position r avec j en position s, on va interdire l'échange inverse, soit l'objet i en position s et l'objet j en position r. Pour ce faire, dans la matrice, à l'indice (i, s), on va inscrire $t + l$, où t est la valeur de l'itération courante et l une valeur fixée. Même chose pour (j, r). On interdit donc le fait, si $t' < t + l$ de faire l'échange inverse en regardant dans les cases (i, s) et (j, r).

2.5 Calcule de Fitness

Une astuce pour ne pas devoir calculer la fitness à chaque fois, qui est en $O(n^2)$. On va simplement calculer la différence qui est, elle, en $O(n)$. On peut faire cela car il y a beaucoup d'éléments qui ne dépendent pas d'i et j. Voici donc la formule :

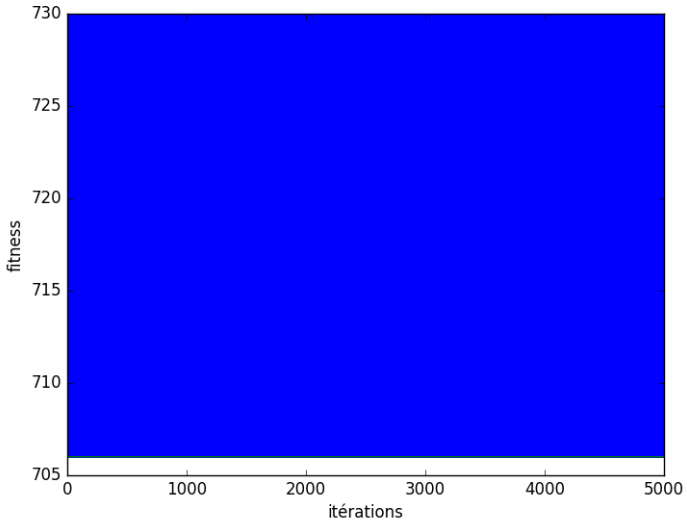
$$\Delta(\psi_i, \psi_j) = 2 \cdot \sum_{k \neq i,j} (w_{j,k} - w_{i,k}(d_{\psi_i,\psi_k} - d_{\psi_j,\psi_k})$$

On a une simple somme, donc le gain calculatoire est conséquent par rapport à une double somme.

3 Résultats

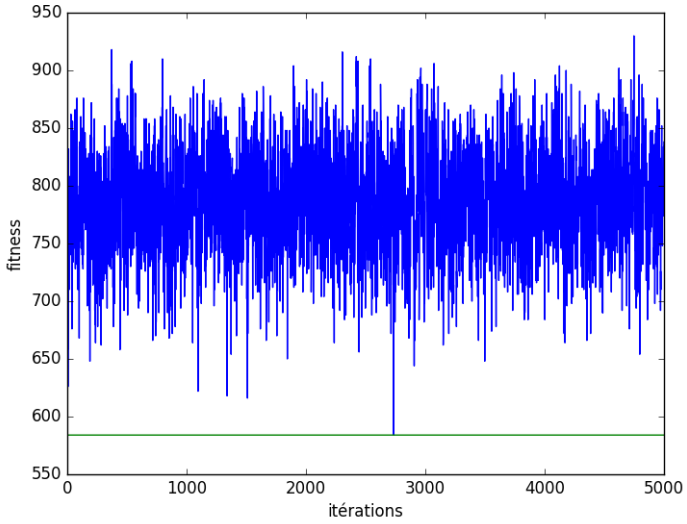
Afin d'avoir une représentation, nous avons dessiné les valeurs de fitness ainsi que le meilleur résultat. Cela permet de voir l'impacte de l sur la recherche.

Pour $l = 1$:



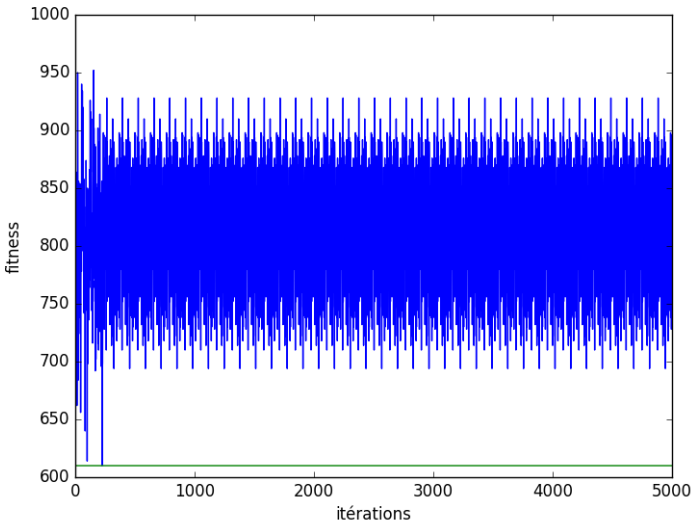
On voit dans ce cas là, que l'algorithme ne prend que deux valeurs de fitness et fait des aller-retour entre. Cela vient du fait, que la taille de la liste tabou vaut 1. Donc il visite une autre valeur, et ensuite il peut revenir sur l'ancienne. Il tombe donc dans le premier minimum local et n'en sort plus. On va voir qu'il est possible de corriger cela en modifiant la valeur de l.

Pour $l = 0.5n$:



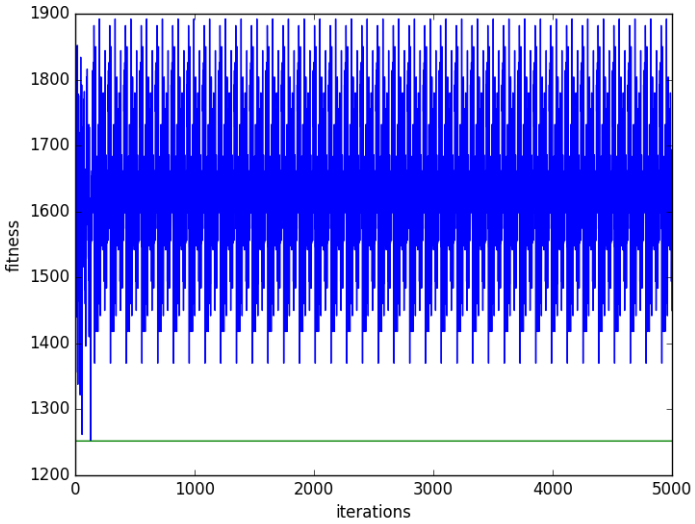
Cette fois ci, malgré qu'il tombe dans certain minima local. La taille de la liste, lui permet d'en sortir pour aller voir autre part dans l'espace de recherche. On remarque également que les fitness varient beaucoup et rapidement. C'est donc une indication sur la difficulté du problème.

Pour $l = 0.9n$:



Ici, l'algorithme va trouver un minimum local, puis va continuer d'explorer l'espace de recherche pour trouver le minimum global (dans cette configuration et pour ce nombre d'itération). Puis il va partir dans une boucle avec un pattern visible.

Les résultats me paraissant étonnants, j'ai généré un nouveau problème de taille $n = 12$, de manière aléatoire et j'ai relancé mon algorithme pour avoir un deuxième graphique :



On voit le même genre de résultat.

Voici les résultats demandés sur dix lancés d'algorithme :

1.dat	best	moyenne	déviation standard
l= 1	626.0	731.28812	39.2576559013
l= 0.5n	544.0	788.65092	43.031624454
l= 0.9n	558.0	811.55036	55.8561665698

On remarque que pour $l = 1$, le fait de tomber dans un minimum local diminue grandement la déviation standard. Car il n'y a que deux valeurs différentes. Pour $l = 0.5n$, il y a une grande différence entre le meilleur et la moyenne. Cela vient du fait que l'exploration continue même une fois qu'un minimum local est trouvé. Même chose pour $l = 0.9n$, sauf que la différence est énorme, du à la boucle qui se crée dans l'exploration.

	best	moyenne	déviation standard
40.dat			
l= 0.5n	18440.0	19598.156	295.173982024
50.dat			
l= 0.5n	29958.0	31371.912	334.404557768
80.dat			
l= 0.5n	78396.0	80086.626	533.669916825
100.dat			
l= 0.5n	122550.0	124811.564	828.609559385

Outre le fait que les valeurs deviennent extrêmement grande, on retrouve le même genre de résultat que pour 1.*dat*. Cela est donc cohérent entre les données.

Un point est à soulever : les performances. Ayant écrit le programme en Python, lorsque n devient grand (pour $n = 40, 50, 80, 100$). Le temps de calcul devient bien trop important. Nous avons tenté de regarder si des optimisations étaient possibles en nous basant sur des outils de *profiling* tel que :

— CPython : pour savoir dans quelle fonction notre programme passait le plus de temps. La réponse est la fonction *delta*.

— Line Profiler : pour savoir combien de temps passer notre programme dans une fonction et quelles lignes demandent plus de calculs que d'autres. Il s'agit de la boucle de calcule du Δ

Malgré cela, il ne nous a pas été possible d'optimiser notre programme. Cependant heureusement qu'on a utilisé le Δ en $O(n)$ et non la fitness en $O(n^2)$. Dans le cadre d'un travail pratique universitaire pour comprendre un algorithme cela peut suffire. Cependant en vue de résoudre un problème concret, il nous faudra un langage plus adapté et avec de meilleures performances.