UNIVERSITÉ DE LORRAINE | UFR MATHÉMATIQUES INFORMATIQUE MÉCANIQUE ET AUTOMATIQUE

Métaheuristique

Frankenstein

USTA Enes - BACHOURIAN Rafaël

Mars 2021

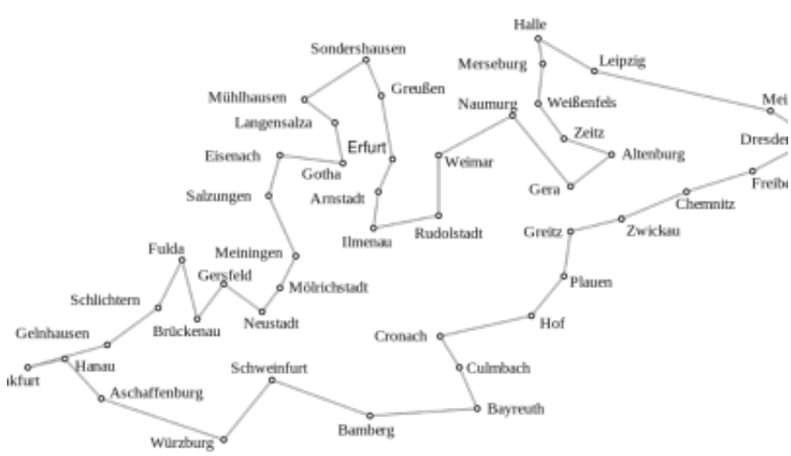
M1 Informatique

Enseignant responsable : M. Alexandre Blansché

Table des matières

1 Introduction

Le problème du voyageur de commerce, ou TSP (Travelling Salesman Problem) est un problème d'optimisation.
Ce problème est le suivant :
Nous disposons d'une liste de villes pour lesquelles, il faut le plus court parcours hamiltonien (qui passe une seule par chaque ville).
Malgré le fait que ce soit un problème simple, il ne peut être résolu dans un temps polynomial.



2 Le projet

Le but de ce projet est de développer un algorithme métaheuristique pour trouver une solution satisfaisante au problème du voyageur de commerce.
Notre projet disposera de 60 secondes pour trouver une solution satisfaisante.
C'est sur la moyenne d'une dizaine d'exécutions que notre projet sera évalué.
Les problèmes qui nous devrons résoudre ici, sont bier127, fnl4461, gr666 et une évaluation surprise1350.

3 Nos heuristiques

3.1 Les opérateurs de mutation

Beaucoup des algorithmes que nous avons étudiés dans cette matière se basent sur le fait de pouvoir modifier une solution déjà existante. Nous allons utiliser ces opérateurs de mutation pour obtenir les solutions modifiées.

Et chaque opérateur n'a pas la même efficacité en fonction des instances du problème.
Il convient donc d'en définir plusieurs, puis de sélectionner ceux qu'on utilisera.

3.1.1 Échange

Le premier opérateur est l'opérateur d'échange. Il échange la position de deux villes aléatoires dans le chemin.
Par exemple :

$$(0\ 2\ 5\ 6\ 7\ 3\ 1\ 4) \Rightarrow (0\ 2\ 1\ 6\ 7\ 3\ 5\ 4)$$

3.1.2 Échange Voisin

Cet opérateur inverse la position de deux villes adjacentes.
Par exemple :

$$(0\ 2\ 5\ 6\ 7\ 3\ 1\ 4) \Rightarrow (0\ 2\ 6\ 5\ 7\ 3\ 1\ 4)$$

3.1.3 Décalage

Cet opérateur change la position d'une ville dans le chemin, mais l'ordre des autres villes reste inchangé.

(6 1 7 2 4 3 5 |) -> (6 1 7 2 | 3 5 4)

3.1.4 Inverse Ordre Zone

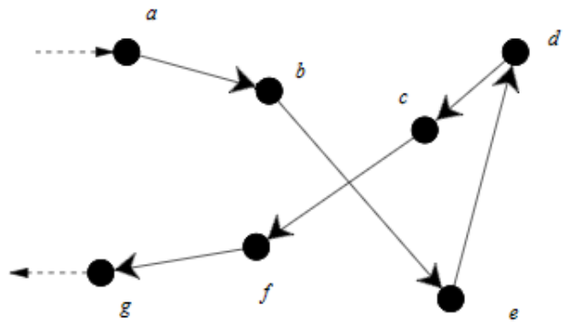
Cet opérateur inverse l'ordre des villes entre deux indices.

(6 1 7 2 | 4 3 5 |) -> (6 1 7 2 | 5 3 4 |)

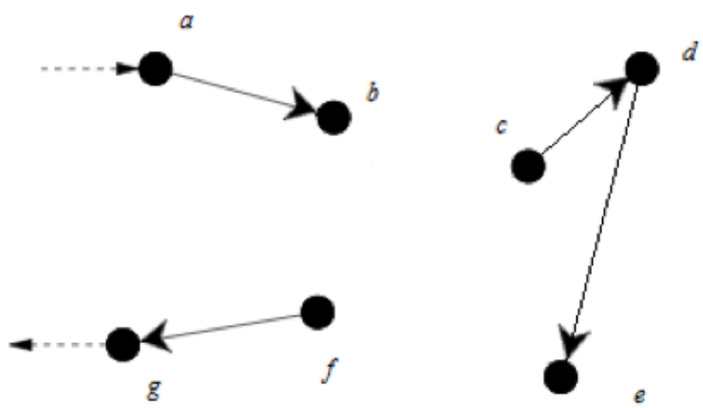
3.2 Optimisation locale

3.2.1 2-Opt

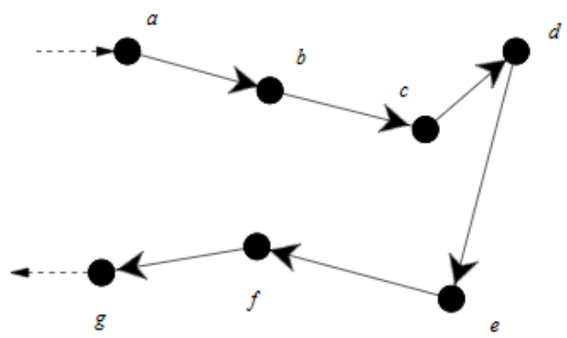
2-Opt est un algorithme d'optimisation qui améliore certains cas d'imprécision dans une solution. L'idée principale de cet algorithme consiste à choisir des routes qui se croisent et de réarranger le graphe pour que les arêtes ne se croisent plus.



Par exemple, pour les nœuds b -> suivant(b) et c -> suivant(c).
On va supprimer les arêtes qui se croisent, et inverser l'ordre du sous chemin entre les deux villes.



Finalement, on relie les noeuds b à c, et suivant(b) à suivant(c).



Malheureusement, cet algorithme est assez lent pour le jeu de données fnl4461.
Et les améliorations qu'il apporte n'en valent pas forcément la chandelle.

3.3 Recuit Simulé

3.3.1 Description

C'est le premier algorithme que nous avons essayé.
Le recuit simulé est un procédé utilisé en métallurgie.
Ce procédé se base principalement sur une température, que nous allons chauffer dès le début à une température très haute.
Plus la température est haute et plus nous serons en capacité d'accepter des solutions dégradantes.

C'est au fur et à mesure des itérations que cette température baisse à l'aide d'un lambda.
C'est lorsque la température sera basse que nous n'accepterons plus de solution dégradante.

3.3.2 L'algorithme

1. Initialisation : Nous effectuons 100 itérations aléatoires, dans notre cas nous avons directement choisi la fonction du voisin le plus proche.
2. De ces 100 itérations, nous en faisons une moyenne des variations des longueurs des solutions.

$$|\overline{\Delta f}|$$

   — Δf : Différence entre la température trouvée et de l'itération précédente
3. Nous calculons ensuite T_0, notre température initiale calculée à partir de cette moyenne :

$$T_0 = \frac{|\overline{\Delta f}|}{ln(p_0)}$$

   — p_0 est la probabilité initiale choisie arbitrairement

   Cette température va baisser toute les 12 et 100 itérations tel que :

$$T_{i+1} = \begin{cases} \lambda T_i & \text{si } \{s,i\} \in E \\ T_i & \text{sinon.} \end{cases}$$

3.4 Ant Colony Optimization

ACO (Ant Colony Optimization) ou optimisation par colonie de fourmis représente le fonctionnement de colonies de fourmis.

Lors d'une itération de la recherche du plus court chemin, les fourmis d'une colonie vont effectuer les étapes suivante, sachant qu'elle seront réparties aléatoirement sur l'ensemble des villes :

1. Choisir une ville, en faisant un compromis entre la distance et des phéromones déposés par des passages précédents de fourmis.
   Notre algorithme utilisera la formule suivante pour calculer la probabilité de choisir l'un des chemins :

$$p_{ij}^k = \begin{cases} \frac{\tau_{ij}(t)^\alpha \eta_{ij}^\beta}{\sum_{l \in j_i^k} \tau_{il}(t)^\alpha \eta_{ij}^\beta} & \text{si } j \in J_i^k \\ 0 & \text{si } j \notin J_i^k \end{cases}$$

   Tel que :
   - t : L'iteration
   - $\tau_{ij}(t)$: La quantité de phéromone entre les villes i et j à l'itération t
   - η_{ij}^β : La visibilité, égale à l'inverse de la distance entre deux villes i et j soit $\frac{1}{d_{ij}}$
   - α : Paramètre contrôlant l'importance des phéromone
   - β : Paramètre contrôlant l'importance de la visibilité
   - j_i^k : La liste des déplacements possibles
2. Après que chaque fourmis ait fait son chemin, une partie des phéromones s'est évaporée.

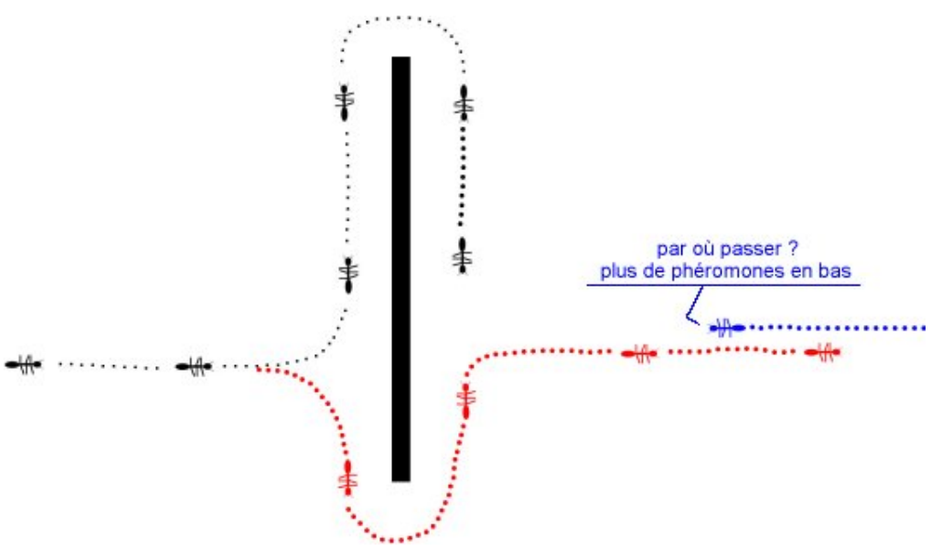
$$\forall i, j \tau_{ij} = \tau_{ij} \times evaporation$$

3. C'est alors que les fourmis parcourent à nouveau leur chemin dans le sens inverse tout en déposant une quantitée de phéromone en fonction de la longueur de leur chemin
   Nous pouvons appeler cela leur contribution et la représenter par :

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L^k(t)} & \text{si } \{i,j\} \in T^k(t) \\ 0 & \text{si } \{i,j\} \notin T^k(t) \end{cases}$$

   Tel que :
   - $\Delta\tau_{ij}^k(t)$: La quantité de phéromone déposée sur chaque arête par la fourmi k
   - Q : Paramètre de contrôle à définir arbitrairement
   - $L^k(t)$: Longueur du trajet effectué par la fourmi k à l'itération t

Exemple d'une décision de fourmi



3.5 Algorithme Génétique

L'algorithme génétique est un algorithme simple qui reprend le concept d'évolution et de sélection présente dans la nature.
Typiquement, une population de solutions est créée au départ, puis à chaque génération, des individus sont sélectionnés pour se reproduire. Certaines des solutions subissent aussi une mutation d'une génération à l'autre.
Les opérateurs de croisements

3.5.1 Croisement PMX

Le croisement PMX (Partially Mapped Crossover) est un croisement de deux parents qui va donner deux enfants.
On choisit un indice aléatoire qui sépare les parents en deux.

(6 1 7 | 2 0 4 3 5)
(0 2 5 | 6 7 3 1 4)

Le premier enfant va être une copie du premier parent dans lequel on va échanger la première ville des deux parents (les villes 0 et 6).

(0 1 7 | 2 6 4 3 5)

Puis le second de chaque parent (le 1 et le 2) etc... et ce jusqu'à l'indice choisit au début.

(0 2 7 | 1 6 4 3 5)

Ce qui donne l'enfant :

(0 1 5 | 2 6 4 3 7)

On reproduit ensuite le processus pour le second parent, ce qui donne le second enfant.

3.5.2 Croisement MOC

Le croisement MOC (Modified Order Crossover)
Un indice aléatoire est choisi afin de séparer en deux parties les gènes de parents. Les villes de la partie gauche du parent 1 sont placées à leur position dans le parent 2 pour le fils 2 et inversement. Ensuite les villes restantes sont placées dans leur ordre d'apparition dans la partie droite du parent 1.

Parent 1 = (1 2 3 | 4 5 6)
Parent 2 = (5 3 1 | 2 6 4)

La partie gauche est placée :

Enfant 1 = (1 * 3 * 5 *)
Enfant 2 = (* 3 1 2 * *)

Les enfants sont complétés avec les villes restantes dans l'ordre :

Enfant 1 = (1 2 3 6 5 4)
Enfant 2 = (4 3 1 2 5 6)

Les deux croisements donnent des résultats très similaires. Mais nous avons choisi MOC car on aime bien le nom.

3.6 Frankenstein : Algorithme génétique grimpeur de colline à colonie de fourmies

3.6.1 Le mélange

Pour ne pas gâcher tout ce temps de travail, nous voulions créer un monstre de foire qui réunirait tout nous algorithmes d'une façon ou d'une autre ! Alors nous avons opté pour celui qui nous donnait les meilleurs résultats, l'algorithme génétique. Nous l'avons peu à peu modifié en y ajoutant des morceaux d'autres algorithmes.

D'abord, nous avons initialisé la population avec les résultats la colonie de fourmis sur quelques itérations.
Notre algorithme génétique était bon, et a trouvé des résultats satisfaisants rapidement, et ne restait pas bloqué dans de très mauvais optimums locaux, mais avait du mal à atteindre les dernières petites optimisations nécessaires pour nous. Une parfait occasion pour utiliser le recuit simulé.

Nous avons eu l'idée de l'exécuter au bout de 100 générations sans améliorations, puis toutes les 1000 générations consécutives si le recuit n'a toujours pas trouvé d'améliorations. Cela permettrait en quelque sorte d'alterner entre une phase d'exploration avec l'algorithme génétique, puis une phase d'exploitation avec le recuit.
Finalement, nous nous sommes rendu compte que l'on avait pas besoin de la partie exploration du recuit, puisque l'exploration est déjà faite par l'algorithme génétique. Nous l'avons donc remplacé par un Hill Climbing.

3.6.2 Le choix des mutations

Le choix des opérateurs de mutations est très dépendant de l'instance et de la taille du problème. C'est pourquoi nous avons décider de tous les utiliser, mais à bon escient !
Chaque opérateur a un score qui est proportionnellement augmenté lorsqu'il contribue à une amélioration.
L'algorithme va se "rappeler" des opérateurs qui ont le plus amélioré la solution. Ils vont donc être sélectionnés plus souvent si ils ont un score plus élevé.
Il y a aussi un facteur d'oubli, qui permet de diversifier les opérateurs quand ils n'apportent plus d'améliorations.

4 Résultats

Nos tests finaux, effectués sur 10 runs et 60 secondes nous ont donné ces résultats.

-	bier127	fnl4461	gr666
Optimum	118282	182566	2943,58
Frankenstein	120 315,01	222 259,20	3 317,94
Frankenstein (Recuit)	121 563,73	222 482,90	3 349,39
Colonie de fourmis	123 470,59	222 094,57	3 640,93
Recuit simulé	126 470,68	226 503,34	3 334,01

5 Conclusion

Le problème du voyageur de commerce est un problème qui nous a été présenté comme très difficile à résoudre, et cette matière a été l'occasion de s'y intéresser.

Ce projet a été intéressant pour nous, il nous a permis de découvrir l'efficacité des algorithmes métaheuristiques pour ce problème et de les comprendre plus en détail. Nous avons programmé des algorithmes classiques de métaheuristique, puis nous y avons ajouté notre touche d'originalité. Et ces modifications que nous avons apportée ont été utiles puisque c'est cette variante qui a été le plus efficace.

Nous pouvons dire que nous avons apporté notre pierre à l'édifice, même s'il ne faut pas oublier que nous ne sommes que des nains sur les épaules de géants.

Références

[1] *Méta-heuristiques d'optimisation par colonie de fourmis*
wikimemoires.net/2014/02/meta-heuristiques-optimisation-colonie-fourmis/

[2] *Ant Colony Optimization with a Java Example*
www.baeldung.com/java-ant-colony-optimization

[3] *Lin-Kernighan heuristic*
akira.ruc.dk/ keld/research/LKH/

[4] *Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation*
user.ceng.metu.edu.tr/ ucoluk/research/publications/tspnew.pdf

[5] *Je Descends de la Colline à Cheval, Mathis Saillot 2020*