Le grimpeur qu'a pas peur 🧗

Gwenaël Kreutner, Théo Meyer

Mars 2022

Table des matières

1 Introduction **2**

2 Les opérateurs de mutations **2**
2.1 Inverse Mutation . . . 2
2.2 Twors Mutation . . . 3
2.3 Move Mutation . . . 3
2.4 Left Shift Mutation . . . 3
2.5 2-Opt Mutation . . . 3

3 Les algorithmes implémentés **3**
3.1 Hill Climbing . . . 4
3.1.1 Description . . . 4
3.1.2 Ses limites . . . 4
3.2 Recuit Simulé . . . 4
3.2.1 Description . . . 4
3.2.2 Amélioration . . . 4
3.3 GRASP . . . 5
3.3.1 Description . . . 5
3.4 Fourmis . . . 5
3.4.1 Description . . . 5
3.4.2 Ses limites . . . 5

4 Notre algorithme : Métagueule **5**
4.1 Algorithme évolutionnaire . . . 5
4.2 Stochastic Hill Climbing . . . 6
4.3 2-Opt . . . 6

5 Notre algorithme final (vraiment) : Le grimpeur qu'a pas peur **7**

6 Résultats **8**

7 Conclusion **8**

Références **9**

1 Introduction

Dans le cadre de ce projet, il nous a été demandé de développer un algorithme métaheuristique et stochastique pour trouver une solution au problème du voyageur de commerce. L'objectif est de créer un modèle de construction de chemins qui puisse avoir le meilleur score possible, de sorte qu'il soit capable de se rapprocher des optima connus.

Le problème du **voyageur de commerce** (aussi connu sous son nom anglais de « Travelling Salesman Problem » ou *TSP*), est un problème de parcours de graphes connu. Il faut trouver le chemin le plus court parcourant toutes les villes une unique fois, et en revenant ensuite à la ville de départ.
L'entrée du problème est la plupart du temps un ensemble de villes, desquelles on considère qu'elles forment un graphe complet. Ici, nous travaillons sur le problème dans sa version symétrique, c'est-à-dire que le chemin de A vers B fait la même distance que celui de B vers A.

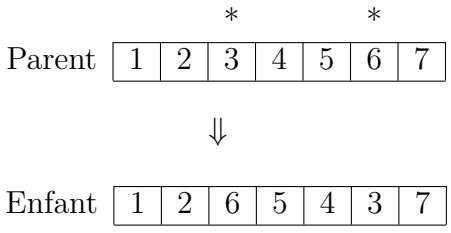
Nous aborderons, dans ce rapport, les différentes techniques métaheuristiques employées afin d'établir un algorithme le plus efficace possible, et donc de répondre à la problématique posée.

2 Les opérateurs de mutations

Les opérateurs de mutations évitent d'établir des populations uniformes incapables d'évoluer. Ils permettent de modifier l'ordre des villes pour un chemin donné. Nous définirons les 5 opérateurs de mutation qui produisent un enfant unique avec un seul parent contrairement aux opérateurs de croisement qui ont besoin de plusieurs parents. Nous avons utilisé ces opérateurs dans la plupart de nos algorithmes afin d'améliorer l'exploration pour trouver une solution optimale. Les 2 premiers proviennent de Bourazza (2006).

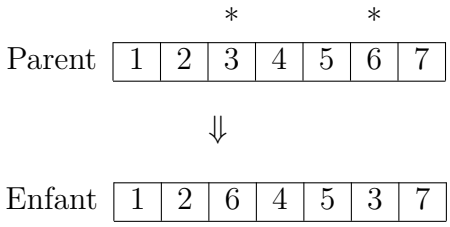
2.1 Inverse Mutation

Cet opérateur échange une séquence de villes entre deux positions choisies aléatoirement. L'ordre des villes sera inversé entre ces deux intervalles.



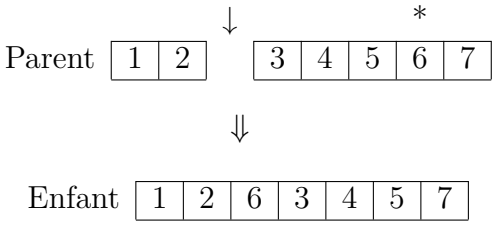
2.2 Twors Mutation

Cet opérateur échange deux positions de villes choisies aléatoirement.



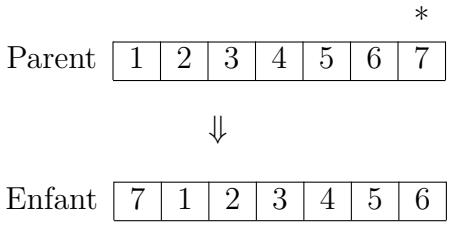
2.3 Move Mutation

Cet opérateur déplace un élément aléatoire et on décale les autres en conséquences.



2.4 Left Shift Mutation

Cet opérateur cycle les villes. Cela n'affecte pas la solution avec un effet direct (la qualité de la solution est exactement la même), mais cela peut permettre d'explorer des solutions différentes dans les itérations suivantes (avec des mutations, croisements...).



2.5 2-Opt Mutation

Cette mutation est à la base de l'algorithme de recherche locale 2-Opt. C'est semblable à la mutation IM, mais les bornes ne sont pas aléatoires.

3 Les algorithmes implémentés

Pour ce projet, nous avons implémenté plusieurs algorithmes métaheuristiques que nous avons vu en cours. Pour la plupart, nous avons appliqué des améliorations que nous allons voir dans cette section.

3.1 Hill Climbing

3.1.1 Description

L'algorithme Hill Climbing essaie de trouver la meilleure solution du problème du voyageur de commerce en commençant par une solution aléatoire, puis en générant ses voisins : des solutions qui ne diffèrent que légèrement de l'actuelle. Si le meilleur de ces voisins est meilleur c'est-à-dire plus court que l'actuel, on remplace la solution actuelle par cette meilleure solution. On répète par la suite ce motif en créant à nouveau des voisins.

3.1.2 Ses limites

L'algorithme est assez simple, mais on ne trouve toujours pas la meilleure solution. En effet on peut rester coincé dans un maximum local, c'est-à-dire à un endroit où la solution actuelle n'est pas la meilleure solution au problème, mais où aucun des voisins directs de la solution actuelle n'est meilleur que la solution actuelle.

3.2 Recuit Simulé

3.2.1 Description

Le recuit simulé a reçu ce nom par analogie avec le « processus de recuit » en thermodynamique, en particulier avec la façon dont le métal est chauffé puis refroidi progressivement afin que ses particules atteignent l'état d'énergie minimale. Ensuite le but de cet algorithme est de chercher aléatoirement une fonction objectif qui caractérise principalement le problème d'optimisation combinatoire. L'avantage du recuit simulé par rapport aux algorithmes est la capacité d'éviter d'être piégé dans les minima locaux. L'algorithme ne rejette toujours pas les changements qui diminuent la fonction objectif mais aussi les changements qui augmentent la fonction objectif en fonction de sa fonction de probabilité :

$$P = \exp\left(\frac{-\Delta f}{T}\right)$$

où T est le paramètre de contrôle (analogie à la température) et Δf est la variation de la fonction objectif.

3.2.2 Amélioration

Il y a deux techniques pour baisser la température, dont une qui varie selon des paliers de température avec les transformations tentées et acceptées qui est censée améliorer notre algorithme. Mais après implémentation de celle-ci, notre algorithme était moins performant, nous sommes donc restés sur la loi de décroissance géométrique.

3.3 GRASP

3.3.1 Description

GRASP est un algorithme appliqué avec succès à de nombreux problèmes d'optimisation. Il consiste en un processus itératif en deux phases. La première phase d'une itération GRASP est la phase de construction, dans laquelle une solution aléatoire gloutonne est construite (exploration). Comme cette solution n'est pas garantie d'être localement optimale, une recherche locale est effectuée dans la deuxième phase (exploitation). Ce processus itératif est répété jusqu'à ce qu'un critère de terminaison soit satisfait et la meilleure solution trouvée sur toutes les itérations est prise comme résultat. Nous nous sommes pas attardés sur cet algorithme, car les résultats de celui-ci étaient trop faibles comparé aux autres.

3.4 Fourmis

3.4.1 Description

L'algorithme Ant Colony Optimization s'inspire du comportement de recherche de nourriture des fourmis. Le comportement des fourmis est contrôlé par deux paramètres principaux : α pour l'attractivité de la phéromone pour la fourmi et β pour la capacité d'exploration de la fourmi. Si α est très grand, les phéromones laissées par les fourmis précédentes dans un certain chemin seront jugées très attractives, faisant que la plupart des fourmis détournent leur chemin vers une seule route (exploitation), si β est grand, les fourmis sont plus indépendantes pour trouver le meilleur chemin (exploration).

3.4.2 Ses limites

Plus la taille de l'instance du nombre de villes est grande, plus notre algorithme prend du temps à créer un chemin pour une fourmi. Ce qui est problématique pour nos grandes instances dont **fnl4461** et **surprise1350**. Nous ne nous sommes donc pas attardé sur cette solution.

4 Notre algorithme : Métagueule

Nous avons basé notre agent sur un mélange de recherche locale avec un « Stochastic Hill Climbing » et un opérateur de croisement. Le concept est détaillé dans Katayama et al. (2000). Le même opérateur de recherche locale est utilisé, mais nous avons implémenté un croisement différent.

4.1 Algorithme évolutionnaire

Notre algorithme final est basé sur le principe classique d'algorithmes évolutionnaires. Il y a plusieurs phases, que nous avons adaptées spécifiquement au problème :

- *Génération de la population initiale* : nous générons la moitié de la population par construction gloutonne, et l'autre moitié de façon aléatoire.
- *Évaluation* : nous couplons chaque solution avec son score.
- *Sélection* : nous trions la population selon le score.
- *Production de la nouvelle génération* : nous générons la nouvelle population par une combinaison de 3 procédés :
  - *Élitisme* : on garde les meilleurs 10% de la génération précédente.
  - *Mutation* : on sélectionne des individus (les meilleurs ont plus de chances d'être sélectionnés), et on les mute selon la méthode de recherche locale SHC (voir section 4.2).
  - *Croisement* : on fait des tournois de 8 individus, et on sélectionne les 2 meilleurs pour effectuer un croisement et générer 2 enfants. Nous avons utilisé l'opérateur de croisement classique de TSPCrossing.
  - *Aléatoire* : on ajoute 1 individu complètement au hasard, on espère pouvoir sortir d'un optimum local de cette manière (🙏).

Dans notre cas, nous avons décidé de travailler avec des populations d'un nombre d'individus variable, car les résultats que nous obtenons changent en fonction de la taille de l'instance. D'après l'expérience, les instances contenant beaucoup de villes bénéficient mieux d'une population moins dense.

4.2 Stochastic Hill Climbing

C'est un algorithme de recherche locale (décrit dans l'algorithme 1), avec une part d'incertitude pour laisser la possibilité de jouer des coups moins bons (Katayama et al., 2000). Il s'agit de répéter une mutation tant que la solution s'améliore. La mutation la plus citée dans la littérature est la mutation **2-Opt**. C'est également la mutation qui nous a donné les meilleurs résultats.

Le but de cette recherche locale est que l'on mute un chemin et que l'on continue tant que cela améliore la solution.

4.3 2-Opt

Cette mutation permet de « décroiser » des arêtes qui s'entrecouperaient (voir figure 1). Pour réaliser cette mutation, il faut trouver un couple d'arêtes qui s'entrecroisent, les supprimer puis recombiner les 2 chemins obtenus d'une manière différente. Il n'existe qu'une manière de les recombiner pour garder une solution valide.
Nous nous sommes basés sur la description qui en est faite dans Jünger et al. (1995).

Il existe une variation nommée 3-Opt qui supprime 3 arcs, mais cela donne alors plus de recombinaisons possibles, et nous n'avons pas eu de bons résultats pour le coût que cela représentait en termes de performance.

Algorithme 1 : Stochastic Hill Climbing

Données : p // *une solution existante et valide*
$p_{sch} \leftarrow 0.1$;
$best \leftarrow p$;
$N \leftarrow n$;
tant que *la solution s'améliore* **faire**
  $x \leftarrow 2OptMutation(p)$;
  si $score(x) < score(best)$ **alors**
    $best \leftarrow x$;
  sinon
    $best \leftarrow x$ avec une probabilité de p_{shc} ;
  fin
fin

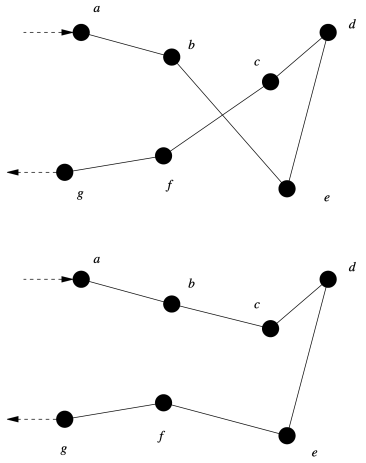


FIGURE 1 – Visualisation d'une mutation 2-Opt

5 Notre algorithme final (vraiment) : Le grimpeur qu'a pas peur 🧗

Finalement, une illumination nous est venue (bien tard), et nous avons essayé de mettre en place la recherche locale via le SHC dans un simple algorithme de Hill Climbing.

Ici, notre agent construit une solution de façon gloutonne, puis le SHC mute cette solution (via la mutation 2-Opt), et ce jusqu'à stagnation de la solution.

Il s'avère que les résultats ont tous été meilleurs que pour notre agent Métagueule. Les résultats sont présentés dans la section suivante.

6 Résultats

Les résultats que nous avons trouvés ont été très satisfaisants à partir du moment où nous avons mis en place le SHC.

On peut apercevoir dans le tableau ci-dessous le récapitulatif des résultats de nos différents algorithmes implémentés. Nous pouvons constater que l'algorithme génétique amélioré (Genetic Hill) a obtenu les meilleurs résultats. Le recuit simulé de base a également des résultats assez prometteurs mais par manque de temps, nous n'avons pas pu approfondir nos recherches pour y apporter des améliorations.

Nom	bier127	fnl4461	gr666	surprise1350
Le grimpeur qu'a pas peur	124722	199115	3332	9719
Métagueule	130408	217457	3410	10315
Hill Climbing	133970	222008	3835	11008
Simulated Annealing	135717	225502	3794	11733
Ant Colony Optimization	372127	$+\infty$	31155	$+\infty$
Grasp	545128	8168263	55000	881838

Table 1 – Comparatif des résultats des différents algorithmes

Nous sommes globalement satisfaits de nos résultats, mais nous pensons que nous aurions pu avoir de meilleurs résultats sur Métagueule en implémentant un opérateur de croisement plus performant, tels que ceux basés sur des « subtours » (Katayama et al., 2000 ; Sosk & Ahn, 2003), ceux basés sur le fait de garder des sous-parties intéressantes (Lu et al., 2017).

Nous aurions également pu tester l'efficacité de la « version généralisée » de 2-Opt, *l'heuristique de Lin-Kernighan* (Karapetyan & Gutin, 2011).

7 Conclusion

Ce projet a été un véritable challenge pour nous, et en même temps très enrichissant. Nous avons pu explorer des possibilités que nous trouvions intéressantes, notamment les nombreux opérateurs de mutations (qui au final se sont avérés moins bons que la mutation 2-Opt), et toute la partie algorithme évolutionnaire. Et puis c'est finalement un algorithme plus simple qui nous a donné les meilleurs résultats en dernière minute, avec la recherche locale qui est très puissante.

Notre algorithme n'est pas parfait et pas non plus le plus élaboré, mais il donne des résultats corrects tout en nous rappelant parfois que les solutions les plus simples sont souvent de bonnes.

Références

Bourazza, S. (2006). *Variantes d'algorithmes génétiques appliquéees aux problèmes d'ordonnancement* (Theses) [24/01/07]. Université du Havre.

Jünger, M., Reinelt, G., & Rinaldi, G. (1995). Chapter 4 The traveling salesman problem. In *Network Models* (p. 225-330). Elsevier. https://doi.org/https://doi.org/10.1016/S0927-0507(05)80121-5

Karapetyan, D., & Gutin, G. (2011). Lin–Kernighan heuristic adaptations for the generalized traveling salesman problem. *European journal of operational research*, *208*(3), 221-232.

Katayama, K., Sakamoto, H., & Narihisa, H. (2000). The efficiency of hybrid mutation genetic algorithm for the travelling salesman problem. *Mathematical and Computer Modelling*, *31*(10), 197-203. https://doi.org/https://doi.org/10.1016/S0895-7177(00)00088-1

Lu, Y., Benlic, U., & Wu, Q. (2017). A Hybrid Dynamic Programming and Memetic Algorithm to the Traveling Salesman Problem with Hotel Selection. *Computers Operations Research*, *90*. https://doi.org/10.1016/j.cor.2017.09.008

Sosk, s. m., & Ahn, B.-H. (2003). New Subtour-Based Crossover Operator for the TSP. *2724*, 1610-1611. https://doi.org/10.1007/3-540-45110-2_49