UNIVERSITÉ DE LORRAINE | UFR MATHÉMATIQUES INFORMATIQUE MÉCANIQUE ET AUTOMATIQUE

Master 1 Informatique
Année 2022-2023

Rapport du projet de Métaheuristique présenté en mars 2023 par

SEMIN Léo et BRUNGARD Luc

en vue de l'obtention du master informatique

Table des matières

1 - Introduction

Le projet à réaliser consiste à développer un algorithme métaheuristique qui permet de trouver une solution acceptable pour le problème de voyageur de commerce.
Le problème du voyageur de commerce étant un problème NP-complet, il faut définir une condition d'arrêt pour évaluer la qualité de l'algorithme développé. Dans notre cas, il s'agit d'une limite de temps, définie à 60 secondes pour l'exécution de l'algorithme. La meilleure solution est retenue après ce laps de temps.

2 - Les ensembles à étudier

Pour ce projet, quatre fichiers contenant les coordonnées des villes ont été fournis :

- big40500, qui est un ensemble de 40500 villes,
- pr8192, un ensemble de 8192 villes,
- tr903, un ensemble de 903 villes,
- th190, un ensemble de 190 villes.

Des fichiers personnels ont été créés à la main en supplément pour tester différentes configurations de villes (exemple : les villes sont toutes alignées).

Pour réaliser des décisions rationnelles, nous avons étudié les tailles des différents problèmes : en effet, l'instance la plus grande que nous avions à notre disposition étant un ensemble de 40500 villes, des réelles questions d'optimisation et de performances algorithmiques ont dû se poser. Plus la taille du problème étudié était grande, plus les fonctions manipulant l'ensemble de villes d'une solution étaient coûteuses. Le temps étant limité, certaines implémentations d'algorithmes ne furent pas possibles en raison de leur trop grande complexité.

3 - Les algorithmes étudiés

Notre premier essai a été de réaliser un recuit simulé car il s'agissait du premier algorithme étudié en cours. Nous avons cependant très vite changé de stratégie et étudié les performances obtenues avec un algorithme génétique.
Lors du développement d'un algorithme génétique, il faut étudier différents paramètres tels que :

- Le nombre d'individus par population
- Le choix de la population initiale
- L'opérateur de croisement entre individus
- L'opérateur de mutation sur un individu
- L'opérateur de remplacement d'une population

Chacun de ces paramètres va être détaillé au cours de ce rapport.

4 - Le nombre d'individus

Comme dit auparavant, la taille d'un problème influence le nombre d'individus. En effet, nous effectuons des opérations de croisement et de mutation qui peuvent être gourmands en complexité, donc plus la taille du problème est grande, moins il y aura d'individus dans une population. Nous prendrons ici un minimum de 10 individus avec un maximum de 50. La formule du nombre d'individus obtenu est : Max(10, 50 - (taille du problème / 1000)). Chaque tranche de 1000 villes retire un individu dans la population.

5 - L'initialisation de la population

La population initiale est une phase très importante des algorithmes génétiques, elle permet d'orienter la recherche de départ. Si elle est mal réalisée, les croisements ne seront pas productifs et la meilleure solution risque de ne pas être intéressante. Nous avons donc testé différentes stratégies d'initialisation.

5.1 - Random

C'est l'algorithme le plus rapide possible : il crée un chemin avec toutes les villes d'un problèmes rangées dans un ordre aléatoire. Sa complexité est en O(n). Ses performances sont cependant très mauvaises. Réaliser une initialisation avec uniquement des solutions aléatoires offre des résultats lamentables, malgré de multiples opérateurs de croisement et de mutation.

5.2 - Greedy

C'est un algorithme glouton qui, en partant d'une ville aléatoire, va créer un chemin en prenant toujours la ville la plus proche. Cet algorithme offre de bien meilleurs résultats que l'algorithme random mais est plus gourmand en temps de calcul.
Nous avons essayé de créer une population initiale avec différent nombre d'individus générés par cet algorithme:

- initialiser une population entière avec des solutions gloutonnes
- initialiser une population avec 1 solution gloutonne et le reste de solutions aléatoires
- initialiser une population avec 2 solutions gloutonnes et le reste de solutions aléatoires

Après observations et de nombreux essais, nous avons conclu que, après 1 individu généré avec l'algorithme glouton, rajouter plus d'individus générés de la même manière n'améliorait pas vraiment la solution.
Nous avons donc décidé de rester sur l'initialisation de la population avec 1 solution gloutonne et le reste de solutions aléatoires.

5.3 - InitializationHeuristic

5.3.1 - L'algorithme et les améliorations

L'initialisation de l'algorithme consiste à sélectionner 4 villes avec le x le plus grand, le x le plus petit, le y le plus petit et le y le plus grand et de créer un chemin non croisé entre ses 4 villes ainsi que de calculer cette distance.
Une fois ceci fait, on entre dans la boucle **i** qui consiste à prendre une ville **v** non insérée et on essaie de trouver à quel endroit on peut insérer la ville telle que la distance ajoutée est minimisée (figure 1).

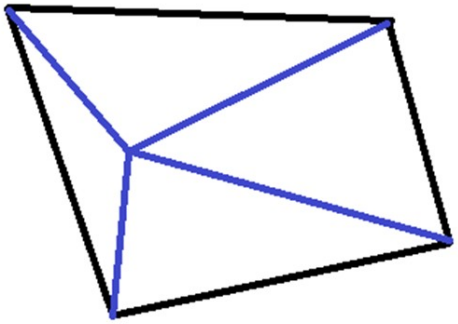

Figure 1. L'itération i avec un chemin de 4 villes

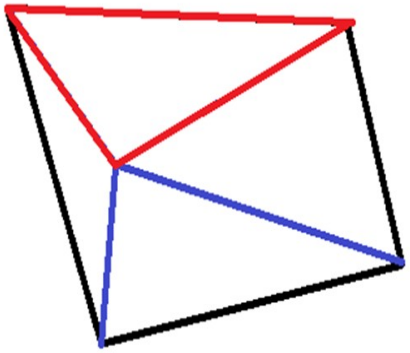

Figure 2. Les distances utilisées pour l'itération j

Pour trouver cet endroit, on itère sur la boucle j qui consiste à simuler une insertion en enlevant la distance entre le sommet **j** et **j+1** et à ajouter les distances **j** à **v** et **v** à **j+1** (figure 2). Une fois ceci fait on compare si la distance est plus courte que celle déjà enregistrée, si oui on la remplace. On peut noter qu'à chaque début d'itération i on initialise la distance à + l'infini pour ne pas avoir de problème.
On continue ceci jusqu'à avoir inséré toutes les villes pour obtenir le chemin final. En termes de complexité on a deux boucles imbriquées qui nous donnent une complexité en $O(n^2)$ bien que **j** itère de 0 à **i** à chaque boucle. De plus, les calculs de distances sont assez gourmands en distances.
Un point qui n'est pas décrit dans l'article original est les cas particuliers. En effet les 4 villes originales peuvent coïncider dans ce cas, il faut supprimer une des deux ce genre de cas arrive dans le cas d'un ensemble de données ressemblant à un triangle ou une ligne.

5.3.2 - Programmation dynamique

Nous avons remarqué que sur les grands jeux de données, l'algorithme mettait assez longtemps à tourner. C'est pourquoi nous avons voulu minimiser le temps de calcul en limitant l'action

gourmandes assez évidentes qui est le calcul de distance qui est parfois fait plusieurs fois pour la même arête.

Il y a deux améliorations possibles :

- D'un côté, si l'on décompose une itération **i**, on peut remarquer que les arêtes créés uniquement pour cette itération (figure 1 - les distances en bleu) sont utilisés deux fois, une première fois pour l'itération **j** et une seconde pour l'itération **j+1**.
- De l'autre côté, on remarque que lors de l'itération **i+1**, on recalcule **i+1** distances de l'itération précédente (figure 1 et 3 - les distances en noire sur la figure 3 avait déjà été calculé lors de l'itération i représenté par la figure 1).

La première méthode nous permet d'économiser au total la somme des termes de 1 à n soit **(n(n+1))/2** distances. La seconde permet elle aussi d'économiser le même nombre de calculs.

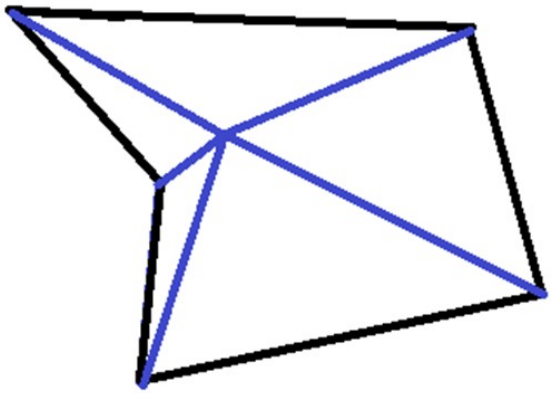

Figure 3. L'itération i+1

C'est pourquoi nous avons gardé en mémoire ces distances dans une variable pour le premier point et dans une table hachage pour le second point. L'optimisation se fait énormément ressentir pour les problèmes de grandes tailles avec en moyenne 5,5 secondes pour le problèmes à 40500 villes.

	big40500	pr8192	tr903	th190
Avec	21400	867	10	8
	24657	958	10	2
	26210	927	11	1
	21419	794	12	1
	29361	888	10	1
	35087	884	12	1
	36109	880	13	1
	35759	867	13	1
	34764	872	11	1
	35894	894	11	1
Sans	24576	889	10	10
	35669	810	11	2
	37564	1027	13	1
	36674	970	14	1
	37301	925	12	1
	38467	921	12	1
	41706	982	11	1
	36388	910	11	1
	33604	906	11	1
	33282	917	11	1
Moyenne avec	30066	883,1	11,3	1,8
Moyenne sans	35523,1	925,7	11,6	2

Figure 4. Temps d'exécution de l'algorithme en millisecondes sur les différents jeux de données

Un axe que l'on peut encore optimiser mais qui est moins significatif que la distance est l'insertion qui est faite de droite à gauche donc dans le pire des cas on déplace N villes. Malheureusement nous n'avons pas eu le temps d'optimiser cet algorithme.

5.3 - Composition finale

Pour finir l'initialisation nous avons choisi de composer notre génération d'un chemin composé par l'algorithme de la section précédente et de remplir le reste par des chemins créés aux hasards car aucunes différences ne se montrait avec l'algorithme Greedy mise à part le temps.

6 - L'opérateur de croisement

Une part importante des algorithmes génétiques est de croiser des individus afin d'en tirer le meilleur et ainsi de générer des populations de plus en plus performantes. Les opérateurs de croisement participent à l'exploitation d'une solution.

Nous avons réalisé un algorithme de croisement un point :

- choix aléatoire d'un point de coupure
- deux enfants sont générés à partir des deux parents

La technique de création des enfants est la suivante :
Notons les parents P1 et P2, et les enfants E1 et E2.
A partir du point de coupure décidé, E1 va prendre toutes les villes de P1 se situant avant ce point de rupture et E2 va prendre toutes les villes de P2 se situant avant ce point de rupture. Le point de rupture est le même pour les deux parents. On rajoute par la suite à E1 les villes manquantes depuis P2 et à E2 les villes manquantes depuis P1.

Les performances sont inconnues mais n'ont pas l'air concluantes. Un opérateur de croisement trouvé dans les articles de recherche pourrait être plus intéressant.

7 - Les opérateurs de mutation

Les opérateurs de mutation permettent l'exploration de l'ensemble de données en réalisant une opération sur un individu pour donner une toute nouvelle solution, inspirée cependant de la précédente.

7.1 - Random

Le premier opérateur de mutation à avoir été implémenté est l'opérateur de mutation aléatoire : il prend deux villes au hasard dans l'ensemble et les inverse. Cet opérateur a grandement démontré sa capacité à être inefficace. Nous l'avons donc très vite écarté.

7.2 - RSM

Cet opérateur prend deux villes au hasard dans un chemin et inverse toutes les villes entre ces deux bornes.

7.3 - RemoveSharp

C'est un algorithme trouvé dans un document de recherche, il consiste à prendre une ville, et étudier si la longueur du chemin est améliorée si on change cette ville de place de manière itérative. On retient la position qui a généré la meilleure solution. Réaliser cette étape pour toutes les villes dans le chemin et un nouveau chemin optimisé est généré. Nous n'avons pas eu le temps d'implémenter cet algorithme entièrement mais il serait intéressant d'étudier ses performances sur des problèmes de taille moyenne ou petite (pr8192, th190, …)

8 - Remplacement générationnel

8.1 - Remplacement basique

Pour le premier remplacement que nous avions fait, nous gardions 10% des meilleurs de la génération précédente. Puis nous avons ensuite ajouté environ 60% en faisant un croisement entre 2 individus tirés au hasard. Pour finir on remplissait le reste par des individus mutés, toujours tirés au hasard.

8.2 - Remplacement probabiliste

Nous avons tout d'abord défini un rang pour chaque individu d'une population. Pour chaque individu, ce rang est calculé de la manière suivante : évaluation de l'individu /des évaluations des individus.
Ensuite, pour chaque individu, on génère un nombre aléatoire et on le multiplie par le rang obtenu et on retient les deux meilleures valeurs.
Ainsi, avec cette solution, chaque individu à une probabilité d'être sélectionné mais on favorise les individus ayant un meilleur rang pour garder de l'élitisme dans notre sélection.

9 - Résultats

	SeminBrungard
big40500	43940.318589837014
pr8192	16382.0
th190	242.54028895780783
tr903	4606.086502236735

10 - Conclusion

Nous avons, à travers ce projet, tenté plusieurs approches qui n'ont pas toujours fonctionné, mais nous avons étudié et combiné différentes techniques, jusqu'à en trouver une offrant des résultats satisfaisants.
La plus grande complexité de ce projet fut de trouver des algorithmes efficaces dans des temps raisonnables.

11 - Références

G. ANDAL JAYALAKSHMI, S. SATHIAMOORTHY, and R. RAJARAM (2001). A HYBRID GENETIC ALGORITHM - A NEW APPROACH TO SOLVE TRAVELING SALESMAN PROBLEM (p 339-355). https://www.worldscientific.com/doi/10.1142/S1465876301000350

Abid Hussain, Yousaf Shad Muhammad, M. Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry & Showkat Gani (2017). Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5676484/

ABDOUN Otman, ABOUCHABAKA Jaafar & TAJANI Chakir (2011). Analyse des performances d'opérateurs de mutation génétique à la résolution du Problème de Voyageur de Commerce. https://www.researchgate.net/publication/282733130_Analyse_des_performances_d'operateurs_de_mutation_genetique_a_la_resolution_du_Probleme_de_Voyageur_de_Commerce

Göktürk Üçoluk. (2002). Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation. https://www.researchgate.net/publication/245746380_Genetic_Algorithm_Solution_of_the_TSP_Avoiding_Special_Crossover_and_Mutation