Problème du voyageur de commerce

Clémence COUSIN
Morgane FIOT

20 juin 2021

1 Introduction

Le but de ce rapport est de présenter notre travail sur la résolution du problème du voyageur de commerce en utilisant des algorithmes génétiques et Hill-Climbing (et ses variantes).

Commençons par rappeler ce qu'est le problème du voyageur de commerce.

Imaginons, un commerçant se déplaçant de ville en ville afin d'écouler sa marchandise. Ce dernier a une liste de villes définie à visiter. Il connaît la distance entre chaque ville et ne doit passer qu'une seule fois par chacune d'entre elles.

Notre commerçant souhaite donc organiser sa tournée de manière à respecter les contraintes ci-dessus tout en minimisant la distance qu'il doit parcourir. Notre but est donc de lui proposer un chemin, un itinéraire répondant à son besoin. Pour résoudre ce problème, nous avons codé différents algorithmes (génétiques, hill-climbing...). Nous allons maintenant vous présenter, dans un premier temps, comment est structuré notre projet. Puis, nous vous décrirons les algorithmes implémentés et la manière dont nous les avons implémentés. Enfin, nous vous présenterons les résultats obtenus.

2 Pour commencer : comment lancer notre projet ?

Pour lancer notre projet, exécuter la classe nommée *TravelGUI* située dans le package *src/ dauphine.cousinfiot.IATravelingSalesman.graph*. Cette classe vous permettra de lancer une interface graphique et d'exécuter nos algorithmes simplement tout en visualisant les solutions proposées.

Vous pouvez également lancer notre classe de test nommée *TestAlgorithm* et située dans le package *src/dauphine.cousinfiot.IATravelingSalesman.test*. Cette classe vous permettra de voir les tests effectués dans la dernière partie de ce rapport, sans interface graphique (réponses montrées dans la console).

3 Structure du code

3.1 Package architecture

Le package architecture de notre projet contient les différents éléments dont nous avons besoin pour modéliser le problème et le résoudre.

On retrouve tout d'abord la classe *City* qui nous permet de générer des villes aléatoirement. Une ville est caractérisée par des coordonnées x et y. Deux villes sont dites identiques si elles possèdent les mêmes coordonnées. Cette classe contient également une fonction permettant de calculer la distance entre 2 villes.

Ensuite, vient la classe *CityMap*. C'est grâce à cette classe que nous générons et stockons un certain nombre de villes. Il est également prévu de pouvoir créer manuellement un ensemble de villes saisies à la main que nous utilisons principalement dans nos tests pour comparer la performance des différents algorithmes. Cette classe génère à la fois une *ArrayList* gardant en mémoire les différentes villes, mais elle calcule également la distance entre chacune des villes dans une *HashMap*. La clé de cette *HashMap* est une paire de villes. Ces paires sont générées via une classe *Pair*.

Pour simplifier la résolution de certains algorithmes, nous avons développé une classe *Travel*. Un voyage donc est un ordre possible pour visiter toutes les villes. Elle récupère en paramètre une variable *CityMap* pour repartir de la bonne liste de villes et peut mélanger l'ordre de ces villes pour générer un nouveau voyage pour le voyageur de commerce. Cette classe permet de calculer la distance à parcourir pour visiter toutes les villes dans l'ordre défini précédemment jusqu'à revenir à la ville de départ. Elle contient également d'autres fonctions que nous verront plus en détail dans le protocole expérimental.

Pour finir, la classe *Population* permet de regrouper un ensemble de *Travel*. Elle permet également d'ordonner les différents objets *Travel* par rapport à la distance totale du voyage. Là encore, nous reviendrons plus en détails sur les fonctionnalités de cette classe dans le protocole expérimental.

3.2 Package algorithm

Nous avons commencé par créer une interface *TravelingSalesmanSolve* qui nous permet d'être sûr de développer toutes les méthodes dont nous avons besoin dans les algorithmes et que les sorties soient au même format. Nous avons donc 3 méthodes en commun :

— une méthode *solve()* qui permet de récupérer la solution de l'algorithme sous forme d'*ArrayList*
— une méthode *getSolution()* pour récupérer la solution sous le format *Travel*
— et une méthode *getIteration()* pour récupérer le nombre d'itérations de chaque algorithme.

Les deux dernières méthodes sont utilisés pour l'interface graphique.

Chaque méthode implémentée correspond à une classe Java. Étant le point central de notre projet, nous les commenterons dans la partie protocole expérimental.

Pour l'algorithme génétique, nous avions besoin d'une fonction permettant de sélectionner aléatoirement un élément d'une liste en fonction de sa probabilité (fonction *fitness*). Pour cela, nous avons réutilisé la classe *RandomSelector* du TP2 vu en classe.

3.3 Package graph

Pour pouvoir représenter le problème du voyageur de commerce de manière plus ludique, nous avons décidé d'implémenter une interface graphique très simple. Cette interface se base sur celle réalisée pour le projet de Java du semestre 1, réalisée par Morgane Fiot et Etienne Cartier (avec son accord).

La classe permettant de lancer l'interface est *TravelGUI*. L'affichage se découpe en deux parties : la partie de gauche affiche les villes représentées par des points, sur la partie de gauche, chaque méthode peut être appelée en appuyant sur le bouton correspondant. Il existe également un bouton tout en haut pour générer un nouvel ensemble de villes. En sélectionnant l'une des méthodes, la solution sera tracée à l'écran, la distance totale de la solution ainsi que le nombre d'itérations nécessaires pour la trouver seront aussi affichées à l'écran.

TravelGUI fait appel à la classe *GridDisplay* pour l'affichage des points. Nous lui passons pour cela en paramètre un objet *CityMap*, qui sera utilisé, de la même manière, pour tous les algorithmes afin de pouvoir comparer visuellement les différentes méthodes.

Le bouton "Set new parameters" appelle une nouvelle instance de *TravelGUI* et génère un nouvel objet *CityMap* en fonction du nombre de villes choisi par l'utilisateur.

Chaque bouton de solution fonctionne sur le même principe : il récupère l'objet *CityMap* et les paramètres qui permettent de dessiner la grille. On fait appel à la méthode *solve()* de chaque méthode afin de dessiner tous les arcs du chemin entre chaque point.

Pour l'algorithme Local Beam Search, il existe une petite spécificité. Avant d'afficher la solution, l'utilisateur doit saisir le nombre d'états à générer pour cette méthode.

3.4 Package test

Dans notre code, nous avons créé un package nommé test afin de regrouper les tests de différents algorithmes. Ce package contient une classe nommée *TestAlgorithm*. Dans cette classe, nous avons en attributs un objet par algorithme. Nous avons ensuite implémenté un constructeur ainsi qu'une fonction *testXX* pour chaque algorithme. Ces fonctions nous permettent de faire appel 100 fois à la méthode *solve()* de chaque algorithme. Nous comparons ensuite le résultat obtenu avec la longueur du chemin optimale et ajoutons dans une arraylist un *0* si nous trouvons le même résultat (à +/- 5 près pour pallier les petites erreurs d'arrondis faites) et un *1* si nous obtenons un résultat différent. En faisant la somme de tous les éléments de l'array obtenu, nous obtenons ainsi le taux d'erreur de notre algorithme.

Dans la fonction *main()*, nous créons respectivement 4, 6, 8 et 10 villes et générons ainsi un graphe dont nous connaissons la solution. Nous faisons ensuite appel à la fonction *testXX()* pour chaque graphe et obtenons ainsi les taux d'erreur pour nos algorithmes sur des graphes à 4, 6, 8 et 10 villes.

Afin de générer des jeux de tests dont nous connaissons la solution, nous avons utilisé un générateur en ligne [1]. Pour que le jeu généré corresponde à la manière dont nous avons créé les villes (désigné avec des coordonnées entières), nous avons multiplié par 10 et arrondi au supérieur

les coordonnées obtenues. Nous avons fait de même avec la longueur du chemin. Pour pallier ces approximations, nous avons donc permis une petite marge d'erreur dans les résultats donnés par nos algorithmes (+/- 5).

4 Implémentation des algorithmes

4.1 Algorithme génétique

A la base de l'algorithme génétique, il nous faut générer une population de k objets *Travel*. La taille de la population est définie par l'utilisateur dans l'interface graphique, la valeur par défaut est de 10.

Pour résoudre notre problème à l'aide de cet algorithme, nous commençons par créer des couples d'objets *Travel* à partir de la population. Ces couples sont constitués aléatoirement en fonction de la fonction *fitness* décrite dans la classe *RandomSelector*[1]. On dit qu'un couple est composé de deux parents.

Chaque couple de parents va produire deux nouveaux itinéraires possibles, que l'on appelle enfants. Ces enfants seront construits via une opération de cross-over sur les parents en se basant sur la méthode *Order Crossover Operator*. Chaque parent va être coupé en deux. L'enfant 1 recevra la partie de gauche du parent 1 et l'enfant 2 recevra la partie de gauche du parent 2. La partie droite de l'enfant 1 sera complétée en fonction des villes qui ne sont pas encore présente dans son itinéraire selon leur ordre d'apparition dans le parent 2 et réciproquement dans l'enfant 2.

$$P1 = (\ 1\ 2\ 3\ |\ 4\ 5\ 6\ )$$
$$P2 = (\ 3\ 6\ 4\ |\ 5\ 2\ 1\ )$$
$$\Downarrow$$
$$E1 = (\ 1\ 2\ 3\ |\ \ldots\ )$$
$$E2 = (\ 3\ 6\ 4\ |\ \ldots\ )$$
$$\Downarrow$$
$$E1 = (\ 1\ 2\ 3\ |\ 6\ 4\ 5\ )$$
$$E2 = (\ 3\ 6\ 4\ |\ 1\ 2\ 5\ )$$

Avant de remplacer notre ancienne population de parents par celle des enfants, nous appliquons un taux de mutation à nos progénitures. Lorsqu'une mutation provient dans un itinéraire, la position de deux villes est échangée.

On applique ensuite un taux d'élitisme, qui permet de remplacer les pires éléments de notre nouvelle population par les meilleurs de l'ancienne.

Nous pouvons maintenant remplacer l'ancienne population par la nouvelle. Pour éviter de rester bloquer sur un minimum local, l'algorithme doit tourner un certain nombre de fois sans trouver d'amélioration du minimum pour s'arrêter. Actuellement, cette limite est de 1 000 itérations, car il n'est pas rare qu'un nouveau chemin plus performant soit trouvé après 500, 700 ou 800 itérations lorsque que nous avons plus de 10 villes à visiter. Pour obtenir des résultats encore plus performants, nous pourrions augmenter cette limite ou la rendre dynamique en fonction du nombre de villes dans notre itinéraire, mais le temps de calcul pourrait alors devenir beaucoup plus long.

4.2 Algorithme simulated annealing

L'algorithme de recuit simulé se base sur un concept de métallurgie qui consiste à augmenter puis réchauffer un métal pour changer ses propriétés. Il fait partie des algorithmes les plus performant que nous avons testés et son temps de calcul reste assez rapide lorsque l'on augmente le nombre de villes dans notre problème. C'est la seule méthode arrivant à calculer quasi instantanément un résultat pour 100 villes.

A l'initialisation de l'algorithme, nous générons un objet *Travel* qui représente un itinéraire aléatoire. A partir de là, chaque itération pour résoudre le problème consistera à faire soit une inversion soit un décalage[1], l'action étant choisie aléatoirement.

L'inversion consiste à choisir une ville au hasard dans notre itinéraire et celle venant juste après (si on choisit la dernière ville de l'itinéraire, la suivante sera la ville de départ). Ensuite, comme le nom de la méthode l'indique, nous inversons leur position.

$$I = (\ 1\ 2\ 3\ 4\ 5\ 6\ )$$

$$On\ choisit\ les\ ville\ 3\ et\ 4.$$

$$\Downarrow$$

$$I_i = (\ 1\ 2\ 4\ 3\ 5\ 6\ )$$

Pour le décalage, on choisit également deux villes qui se suivent et qui forment donc un arc. C'est cet arc qui sera déplacé à une nouvelle position aléatoire dans l'itinéraire et différente de la position de départ.

$$I = (\ 1\ 2\ 3\ 4\ 5\ 6\ )$$

$$On\ décale\ l'arc\ (3;4)\ à\ la\ position\ 4.$$

$$\Downarrow$$

$$I_d = (\ 1\ 2\ 5\ 3\ 4\ 6\ )$$

Après avoir effectué une des deux opérations précédentes, on regarde si cette nouvelle solution est meilleure que la précédente. Si oui, on remplace toujours l'itinéraire précédent par le nouvel itinéraire. Sinon, on calcule un nombre aléatoire entre 0 et 1 k et $a = \frac{\exp^{-(maxDist-minDist)}}{minDist}$. Si $a > k$, on accepte la modification même si elle fait baisser notre score.

Comme pour l'algorithme génétique, notre algorithme s'arrête après 1 000 itérations sans amélioration d'itinéraire.

4.3 Algorithme local beam search

L'algorithme de recherche en réseau ou local beam search, permet de conserver en mémoire k états différents de notre problème. A chaque itération, on génère tous les descendants d'un état k_i. Ensuite, on récupère les k meilleurs éléments créés.

A l'initialisation, l'utilisateur choisi le nombre k d'éléments à conserver à chaque itération. k itinéraires aléatoires sont ensuite générés.

Ensuite, pour chaque itinéraire, nous calculons tous ses descendants. Un descendant étant l'itinéraire parent où l'on inverse deux villes.

$$P = (\ 1\ 2\ 3\ 4\ )$$

$$\Downarrow$$

$$E1 = (\ 2\ 1\ 3\ 4\ )$$

$$E2 = (\ 1\ 3\ 2\ 4\ )$$

$$E3 = (\ 1\ 2\ 4\ 3\ )$$

$$E4 = (\ 4\ 2\ 3\ 1\ )$$

Parmi tous les descendants ainsi générés, on sélectionne les k meilleurs pour remplacer notre ancienne population. On garde en mémoire la meilleure solution trouvée jusqu'ici. Si on trouve dans la descendance, un meilleur itinéraire, on le remplace.

Comme pour les algorithmes précédents, notre algorithme s'arrête après 1 000 itérations sans amélioration d'itinéraire.

4.4 Algorithme Hill-Climbing et ses variantes

L'algorithme Hill-Climbing est un algorithme utilisant les notions de minimum/maximum. Ici, l'algorithme s'arrête lorsque nous avons trouvé un chemin de taille minimale permettant de visiter exactement une fois chaque ville. Le minimum trouvé peut-être un minimum global ou local.

A l'initialisation de l'algorithme, nous générons un objet *Travel*. Cet objet va nous permettre de créer un premier graphe solution aléatoirement. Ensuite, à chaque itération, nous générons les graphes "voisins" du nôtre. Nous échangeons deux arêtes dans le graphe solution trouvé afin de créer un nouveau graphe solution. Puis, nous calculons la longueur, la distance totale parcourue par le voyageur de commerce, de chaque graphe. Dans les voisins trouvés, nous prenons le graphe qui a fait parcourir la plus petite distance au voyageur de commerce et comparons cette distance à celle de notre graphe solution initiale. Si elle est strictement plus petite, le "meilleur voisin" devient

notre nouveau graphe solution et nous continuons l'algorithme. Sinon, nous arrêtons l'algorithme et proposons comme solution le dernier graphe trouvé. Nous utilisons donc ici la méthode *2-opt*.

L'algorithme Hill-Climbing possède plusieurs variantes. Nous en avons implémenté trois : *First choice Hill-Climbing*, *Stochastic Hill-Climbing* et *Random Restart Hill-Climbing*. Ces trois variantes utilisent toutes la méthode *2-opt*, leurs différences viennent de la manière de choisir le graphe à comparer au graphe solution précédemment trouvé.

Dans l'algorithme *First choice Hill-Climbing*, on choisit le premier graphe voisin qui a un itinéraire plus court que celui du graphe solution, même si un autre de ces voisins à une meilleure solution.

Dans l'algorithme *Stochastic Hill-Climbing*, on choisit au hasard l'un des voisins qui a un itinéraire plus court que celui du graphe solution.

L'algorithme *Random Restart Hill-Climbing* consiste à redémarrer plusieurs fois l'algorithme originel *Hill-Climbing* afin de maximiser les chances de trouver le chemin le plus court. En effet, l'algorithme *Hill-Climbing* peut s'arrêter sur un chemin n'étant pas le plus court possible, mais étant un minimum local. En redémarrant plusieurs fois l'algorithme, avec des graphes de départ différents, nous augmentons ainsi nos chances de trouver un minimum global.

Nos hypothèses sont que l'algorithme *Random Restart Hill-Climbing* sera la version la plus performante des algorithmes *Hill-Climbing*. Nous allons confirmer ou infirmer cette hypothèse dans la partie suivante.

Nous nous sommes aidées de ressources pour implémenter cet algorithme [3].

5 Résultats observés

5.1 Présentation des résultats

Nous avons testé nos algorithmes sur des graphes à 4, 6, 8 et 10 villes dont nous connaissions la solution. Voici les résultats que nous avons obtenu :

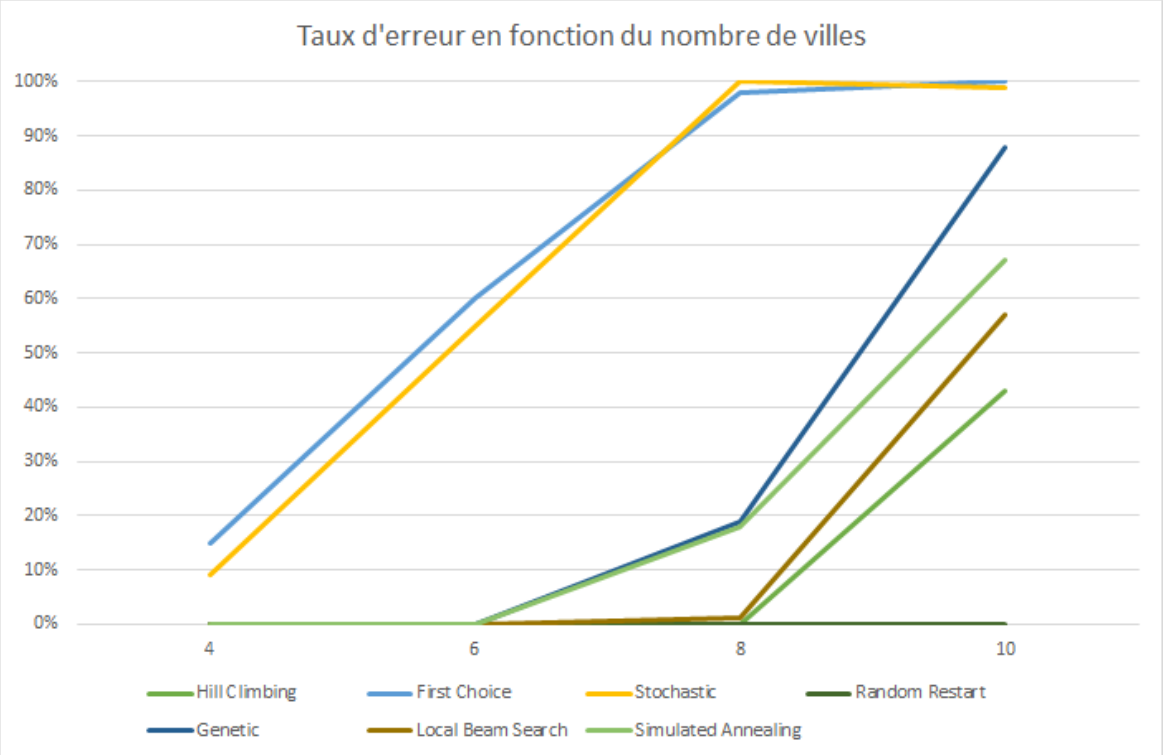

FIGURE 1 – Taux d'erreur obtenus pour les différents algorithmes implémentés.

Nous pouvons remarquer que dès 4 villes, les algorithmes *First Choice Hill-Climbing* et *Stochastic Hill-Climbing* ont un taux d'erreur supérieur aux autres algorithmes (qui ne font pas d'erreur sur un petit graphe). Nous pouvons remarquer le même phénomène lorsque nous augmentons la taille du graphe à 6 villes.

Lorsque nous continuons d'augmenter la taille du graphe (en passant à 8 villes), nous observons que les algorithmes génétiques (*Genetic Algorithm* et *Simulated Annealing*) commettent à leur

tour des erreurs. *First Choice Hill-Climbing* et *Stochastic Hill-Climbing* atteignent même un taux d'erreur de 100% : sur les 100 tests réalisés, nous ne trouvons donc jamais la meilleure solution pour résoudre le problème. Nous pouvons observer que les deux autres algorithmes *Hill-Climbing* implémentés ainsi que l'algorithme *Local Beam Search* ne commettent toujours pas d'erreur. Ils permettent de résoudre le problème de manière optimale à chaque fois.

Lorsque nous passons à 10 villes, seul l'algorithme *Random Restart Hill-Climbing* continue de résoudre le problème de manière optimale à chaque fois. L'algorithme *Hill-Climbing* propose la meilleure solution dans un peu plus de 50% des cas. De la même manière, *Local Beam Search* et *Simulated Annealing* se trompent environ une fois sur deux. Pour l'*algorithme génétique*, on notera une très forte hausse du taux d'erreur avec près de 90% de mauvaise réponse.

Si nous continuions d'augmenter le nombre de villes, en suivant les tendances données par les courbes ci-dessus, nous remarquerions sûrement que l'algorithme *Random Restart Hill-Climbling* reste le plus fiable pour obtenir une solution optimale. Tous les autres algorithmes semblent ici atteindre leur limite pour des graphes contenant entre 8 et 10 villes.

Pour l'exemple ci-dessus, nous avions fixé à 4 le nombre de relances pour l'algorithme *Random Restart Hill-Climbing*. Regardons si les performances de l'algorithme restent aussi bonnes si nous baissons le nombre de relances. Vous trouverez ci-dessous un tableau nous donnant les taux d'erreur de l'algorithme en fonction du nombre de relances.

Nombre de relances	4 villes	6 villes	8 villes	10 villes
2	0	0	1	1
3	0	0	1	1
4	0	0	0	0

TABLE 1 – Evolution du taux d'erreur de l'algorithme *Random Restart Hill-Climbing* en fonction du nombre de restart

Nous n'avons pas traité le cas où nous ne lançons l'algorithme qu'une fois, car cela reviendrait à simplement exécuter l'algorithme *Hill-Climbing*.

Nous pouvons observer que, quel que soit le nombre de relances, l'algorithme ne commet pas d'erreurs pour 4 et 6 villes. Cependant, nous pouvons noter qu'à partir de 8 villes, il faut à minima relancer l'algorithme 4 fois pour obtenir à coup sûr la solution optimale. Nous pouvons donc en déduire que plus nous avons de villes dans notre graphe, plus nous devrons relancer un grand nombre de fois l'algorithme *Hill-Climbing* pour obtenir à chaque fois une solution optimale. Il est donc important d'adapter les paramètres en fonction du graphe à résoudre.

Pour réaliser nos tests, nous avons également dû définir une taille de population de départ pour les algorithmes *génétique* et *Local Beam*. Pour réaliser cette expérience, nous nous sommes basées sur 10 villes, car c'est à ce niveau que l'on observe les taux d'erreur les plus hauts.

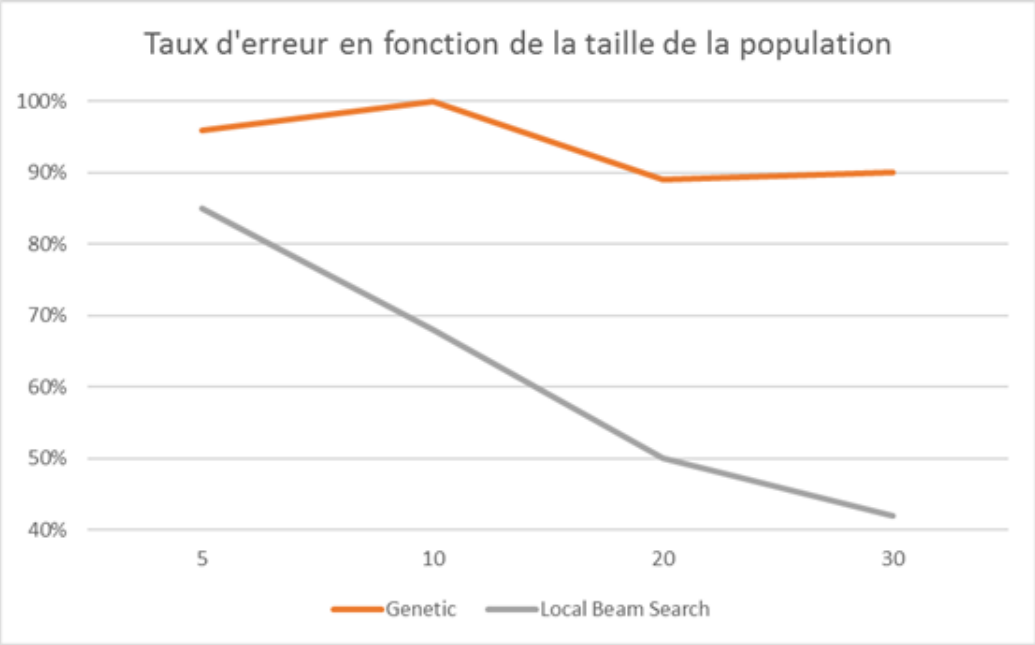

FIGURE 2 – Taux d'erreur obtenus pour différentes tailles de population

Comme nous pouvions nous y attendre, augmenter la taille de la population permet, en règle générale de faire baisser le taux d'erreur. Cela est facilement remarquable pour le *Local Beam Search*. En revanche pour l'*algorithme génétique*, on remarque des résultats qui alternent un peu plus entre augmentation et diminution du taux. Cela peut s'expliquer par le fait que l'*algorithme génétique* est plus sensible à la population de départ que *Local beam search* malgré un taux de mutation aléatoire de 20%.

Dans notre FIGURE 1, nous avons décidé de prendre une taille de population par défaut de 20 individus. Cela nous permet d'obtenir des résultats plus fiables sans pour autant augmenter de beaucoup le temps de calcul. En effet, pour une population de taille 30, il faut plusieurs minutes pour calculer un chemin optimal.

5.2 Analyse des résultats

A travers les exemples que nous avons pu voir ci-dessus, il apparaît que certains algorithmes semblent plus adaptés pour résoudre ce problème que d'autres.

Prenons les algorithmes *Hill-Climbing*. Pourquoi, alors que ces algorithmes sont tous assez proches, certains seraient plus efficaces que d'autres ?

Cela vient tout simplement de la manière de sélectionner le graphe solution. En effet, les algorithmes *First Choice Hill-Climbing* et *Stochastic Hill-Climbing* sélectionnent (quand cela est possible) une meilleure solution que le graphe actuel. Cependant, ils ne sélectionnent pas à chaque fois le meilleur graphe possible, celui avec le chemin le plus court parmi tous les graphes voisins. Nous pouvons donc tomber plus facilement sur un minimum local et non un minimum global. L'algorithme *Hill-Climbing* nous permet donc d'obtenir de meilleurs résultats que ses deux variantes citées précédemment, car nous sélectionnons à chaque fois la meilleure solution possible et augmentons ainsi nos chances de trouver un graphe dont la solution est un minimum global et non un minimum local comme précédemment.

Cependant, l'algorithme *Random Restart Hill-Climbing*, en redémarrant un certain nombre de fois l'algorithme *Hill-Climbing*, permet d'obtenir plusieurs graphes solutions et donc de savoir si le graphe initialement trouvé était la meilleure option possible. En partant d'un graphe initial différent, nous ne suivons pas exactement les mêmes itérations que précédemment et donc nous avons plus de chances de tomber sur un minimum global que local. En calculant intelligemment le nombre de relances nécessaire pour obtenir une solution optimale (selon le nombre de villes de départ notamment), l'algorithme *Random Restart Hill-Climbing* fournit une réponse fiable au problème du voyageur de commerce.

En revanche, si nous considérons un grand nombre de villes de départ (qui nécessiterait donc un grand nombre de relances de l'algorithme *Hill-Climbing*), nous pouvons facilement imaginer que cet algorithme ne sera pas optimal. En effet, d'après ce que nous avons pu étudié ce semestre, l'algorithme *Hill-Climbing* et ses variantes sont des algorithmes gloutons, qui peuvent rapidement être bloqués lorsqu'un minimum local est trouvé ou qu'un plateau est atteint.

Pour l'algorithme génétique, on observe de meilleures performances sur des petits échantillons de villes. En plus d'avoir un taux d'erreur plus élevé, lorsque l'on augmente le nombre de villes, on remarque une augmentation significative du temps de calcul de la solution qui est dû à sa complexité plus importante que d'autres algorithmes. Il est possible qu'en utilisant une fonction de crossover différentes, nous puissions augmenter les performances de notre algorithme. Nous pourrions par exemple utiliser un Cycle Crossover Operator et comparer ses performances avec celui que nous avons implémenté. Nous devons aussi envisager que cet algorithme n'est pas optimal pour résoudre ce genre de problèmes.

L'algorithme *Simulated Annealing* permet d'obtenir des résultats satisfaisant et de manière quasi-instantanée peu importe le nombre de villes. En introduisant d'autres types de modifications, là encore, nous pourrions obtenir des résultats différents.

Le *Local Beam Search* est l'un des algorithmes les plus performants, notamment, car il explore tous les successeurs d'un état. Bien évidemment, plus la population est grande, plus le temps de calcul sera important. Son principal inconvénient est qu'il dépend énormément des individus de départ ce qui peut expliquer son taux d'erreur de plus de 50% au-delà de 10 villes.

Références

[1] Hussain, A. (2017, 25 octobre). *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator*. Hindawi.
`https://www.hindawi.com/journals/cin/2017/7430125/`

[2] Walker, J. (2018, juin). *Simulated Annealing : The Travelling Salesman Problem*. Fourmilab. https://www.fourmilab.ch/documents/travelling/anneal/

[3] Hein de Haan (2008, 8 décembre). *How to Implement the Hill Climbing Algorithm in Python*. Towardsdatascience. https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de/