Antoine LELONG
Elisa PIARALY

ENSAE 1ère année
Année scolaire 2020-2021

Solutions approchées au problème du voyageur de commerce

Projet d'informatique

Table des matières

1 Introduction

Le « Traveling Salesman Problem » ou plutôt le problème du voyageur de commerce est un problème d'optimisation largement répandu en informatique et énoncé pour la première fois par William Rowan Hamilton au 19ème siècle de la façon suivante : « Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur ». L'énoncé paraît simple mais la méthode naturelle qui consisterait à comparer tous les parcours possibles peut s'avérer être très lourde pour un grand nombre de villes. En effet, le nombre de parcours possibles est de (N-1) !/2 pour N villes. Par exemple, pour 25 villes, il s'agirait de faire 3.10^{23} opérations. Les serveurs les plus puissants actuellement permettent de faire 10^{17} opérations par seconde et pourraient résoudre ce problème en près de 860 h.
Ainsi, il convient d'étudier d'autres méthodes de résolution dites « approchées » permettant de trouver un « bon candidat » de chemin, pas forcément optimal, mais en réduisant drastiquement le temps de calcul. Nous nous sommes intéressés à la résolution par l'algorithme de colonies de fourmis puis par un algorithme génétique. Nous exposerons donc ces deux méthodes puis nous partagerons les difficultés que nous avons rencontrées pour établir ces deux algorithmes. Nous comparerons ensuite les performances des différentes pistes explorées dans l'algorithme génétique pour enfin présenter les voies d'améliorations à cette résolution.

2 Description du code

2.1 Approche de la colonie de fourmis

Principe

L'approche de la colonie de fourmi consiste à placer un certain nombre de fourmis dans des villes aléatoire et à suivre leurs parcours individuels. Elles choisissent la prochaine ville qu'elles comptent visiter en fonction de la distance par rapport à leur position actuelle et de la quantité de phéromones déposées par les fourmis précédentes. Ensuite, une fois que chaque fourmi a visité toutes les villes, elles déposent plus où moins de phéromones sur le chemin qu'elles ont parcouru en fonction de la distance qu'elles ont parcouru. Les phéromones des étapes précédentes subsitent, mais s'évaporent graduellement au fil des itérations.

Utilisation du code

L'algorithme décrit précédemment est codé dans la classe *Colonie* qui s'initialise avec l'argument *villes*, un tableau de taille $n \times 2$ contenant les coordonnées des points du problème. Ses attributs notables sont *pref* la matrice des phéromones déposées sur les chemins, *distances* la matrice des distances entre les points afin de faire des économies pendant les calculs de longueur des chemins, *meilleur* qui après au moins une itération contient le chemin de la meilleure fourmi lors de la dernière itération. La méthode qui execute une itération de l'algorithme est *step* qui prend en argument *nbr_fourmis* le nombre de fourmis lancées à cette itération, *dstpwr et prefpwr* les poids relatifs de la distance et des phéromones respectivement dans les décisions des fourmis, et *evaporation* un coefficient entre 0 et 1 par lequel les quantités de phéromones de l'étape précédente sont multipliées.
On peut lancer de nombreuses itérations à la fois avec la méthode *run*, qui prend les mêmes arguments en plus d'un nouvel argument *it* correspondant au nombre d'itérations voulues.

Convergence

Au début, les fourmis explorent des chemins variés et les pistes de phéromones forment une pondération des chemins n'indiquant que des suites de segments privilégiées mais pas de chemin clair. En faisant l'arbitrage entre ces segments, les fourmis finissent par converger sur un seul chemin, et les depots de phéromones sur d'autres segments ne se font plus, rendant impossible l'exploration de nouveaux chemin ce qui garantit la stabilité de l'équilibre trouvé par le système dès lors qu'il en trouve un.

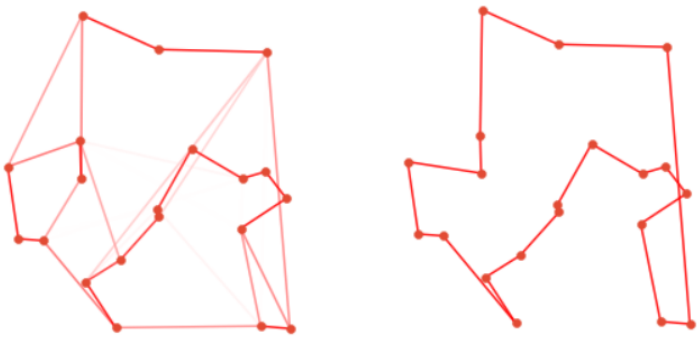

FIGURE 2.1 – Pistes de phéromones après 100 et 1000 itérations

2.2 Algorithme génétique

Principe

L'algorithme génétique implémenté consiste à prendre une population de départ de k individus. Un individu est caractérisé par un gène qui sera l'ordre dans lequel il visite les villes numérotées de 1 à n. Chaque individu se voit attribuer un score en fonction de la longueur de chemin qu'il parcourt. Les individus sont donc classés et les $n_{\text{élites}}$ premiers, sont gardés pour la génération suivante. Celle-ci sera complétée par des croisements d'individus choisis proportionnellement à leur contribution au score total, qui subissent en plus des mutations aléatoires. Les techniques de croisement et mutations sont nombreuses et plusieures ont été testées.Une fois la nouvelle génération créée, il s'agit de réitérer le processus t fois. Les t itérations étant effectuées, l'algorithme retourne la population finale, ainsi que des listes contenant la longueur parcourue par le meilleur individu et la longueur moyenne parcourue par la population à chaque itération.

Utilisation du code

La fonction faisant tourner l'algorithme génétique est *algo_gen*. Elle prend en arguments *villes*, *n_chemin*, *iterations*, *n_elites*, *ordonné*, *taux*, *t*, et *seuil_stag*. Les quatre premiers arguments sont assez explicites à savoir le tableau de coordonnées des villes, le nombre de chemins dans la population, le nombre d'itérations et le nombre d'élites. Le paramètre *ordonné* est un booléen optionnel qui prend par défaut la valeur *True*. S'il vaut *True*, la fonction de croisement *croisement_ordonné* sera utilisée, sinon *croisement_echantillon* sera utilisée. Le paramètre *taux* est un booléen optionel qui prend par défaut la valeur *True*. S'il vaut *True*, la fonction de mutation *mutation_taux* sera utilisée et le paramètre t correspondra au taux de mutation, sinon *mutation_recuit* sera utilisée et le paramètre t correspondra à la température initiale du système. Le paramètre *seuil_stag* permet de définir un seuil de stagnation au delà duquel on arrête l'algorithme : si on dépasse ce nombre d'itération depuis la dernière amélioration du meilleur score, on arrête la boucle. Il prend par défault une valeur infinie, i.e. sauf précision il n'y a pas de seuil de stagnation.

Les fonctions utilisées par *algo_gen* reposent énormément sur la fonction *longueur*, qui permet de calculer la longueur d'un chemin. On utilise en réalité soit la fonction *longueur2* ou *longueurd* selon la dimension du problème, en plaçant la bonne fonction dans la variable *longueur* avant d'exécuter *algo_gen*. La fonction *longueur2* utilise le même algorithme que *longueurd* mais ne nécessite pas la construction d'une liste pour faire des sommes sur chaque dimension puisque la forme du tableau *villes* donnné en argument est connue, ce qui permet un calcul plus rapide qu'en utilisant *longueurd* pour un problème en deux dimensions. La complexité de *longueurd* est en $\mathcal{O}(n \times d)$ où n est le nombre de villes et d la dimension du problème tandis que *longueur2* est en $\mathcal{O}(2n) = \mathcal{O}(n)$.

3 Difficultés rencontrées

Le problème du voyageur de commerce étant un terrain de jeu très visité en informatique, il s'agissait dans un premier temps de sélectionner les méthodes de résolutions et leur variantes qui nous intéressaient. Cependant, la quantité d'informations disponibles sur le sujet nous a parfois mené à nous perdre vers des pistes qui n'étaient pas en lien direct avec notre sujet, comme la théorie des graphes en général ou encore la simulation de déplacement de fourmis. Mais une fois l'objectif défini, il a été plus simple de cibler les informations utiles.
Lors de l'écriture du code de colonies de fourmis, les fonctions principales n'étaient pas complexes à comprendre mais les formaliser pour ensuite les implémenter ont demandé un certain temps pour que chacun de nous deux soit à l'aise avec les notations utilisées.
Certaines erreurs de frappe ou d'indexation ont rythmées cette partie de notre projet, mais en adoptant le réflexe de toujours tester nos fonctions et de mettre des « print() » à chaque étape de notre code, nous a permis de ne pas avancer aveuglément.
Concernant l'implémentation de l'algorithme génétique, les étapes de scoring, de sélection, de croisement et de mutation ont différentes variantes. Nous avons donc décidé de nous partager le travail en deux avec l'un qui implémente l'algorithme génétique avec le croisement par échantillonnage et les mutations par récuit simulé et l'autre avec le croisement ordonné et les mutation par taux.
Cependant, lorsque nous avons voulu comparer nos résultats, nous nous sommes rendu compte que bien que nous ayons implémenté des algorithmes similaires (à savoir génétiques), nos typage n'était pas les mêmes et hautement incompatibles. A titre d'exemple les gènes de l'un étaient l'ordre des indices des villes et de l'autre à : la liste des coordonnées des villes réordonnée. De même pour l'attribution des scores, l'un avait rangé les scores dans un tableau et l'autre dans un dataframe. Ainsi, il a fallu faire des choix de typage globaux les plus simples possible pour fusionner les résultats et comparer les performances de chaque méthode de croisement et mutation, point que nous abordons ci-après. Cette fusion nous a forcé à choisir l'implémentation dans une fonction et non une méthode dans un soucis de simplicité.

4 Comparaison des résultats

4.1 Paramètres de l'algorithme génétique

Fonctions de croisement

Nous avons implémenté des fonctions de croisement très différentes : par « echantillonnage » et de façon « ordonnée ». Ces deux méthodes viennent de conceptions différentes de créer une permutation à partir de deux autres. L'échantillonnage cherche à conserver l'ordre d'apparition de certain points du chemin, tandis que l'ordonné cherche à conserver un bloc du chemin tel quel.
Empiriquement, nous avons remarqué que leurs performances sont proches pour un très faible nombre de villes, mais dès qu'il augmente beaucoup (20 ou plus) le croisement ordonné est bien meilleur. En effet, le croisement échantillonné est plus instable et ne conserve pas les suites de segments mais des ordres de points, qui ne contiennent pas l'information clé du problème.

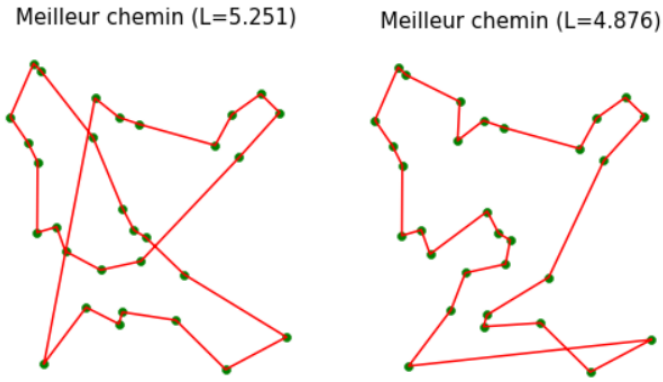

FIGURE 4.1 – Résultats de l'algorithme génétique pour 30 villes avec croisements echantillonés (gauche) et ordonnées (droite)

Fonctions de mutation

De même nous avons choisi deux fonctions de mutation différentes, l'une par taux de mutation et l'autre par recuit simulé. La première pour des taux de mutation très faible (t = 0.01) donne d'excellents résultats mais ralentit la convergence à mesure que le taux diminue. Elle devient également beaucoup moins fiable dès que le taux augmente un peu trop (t = 0.05), mais peu parfois produire des solutions bien meilleures. En effet, dès que le taux est un peu trop élevé, les transpositions positives ont trop de chance d'êtres cachées par des transpositions à l'effet inverse lors de la mutation, résultant en un mutant inintéressant. A l'inverse si le taux est trop faible, il y a trop peu de mutations et on

explore pas assez de possibilités.
La méthode de recuit simulé n'accepte les mutations que dans deux cas : si elle est positive ou dans le cas ou elle est négative, avec une probabilité qui diminue avec la perte et quand la température du système baisse. Ces tests ont un grand coût en temps, ce qui oblige à ne faire des mutations que sur un individu choisi aléatoirement afin de garder un temps d'exécution de l'ordre de celui de la première méthode. On explore donc moins de possibilités, et la convergence est plus lente, mais les résultats restent acceptables. Aussi, l'ensemble des individus de chaque générations ont tendance à rester proches des meilleurs puisqu'avec moins de mutation, la selection determine d'avantage l'apparence de la génération suivante. Partir d'une température haute ne fait que ralentir la convergence (on accepte plus de mauvaises mutations). De manière générale, la première méthode semble plus efficace, tant par sa capacité à explorer plus de mutations en autant de temps de calcul que par ses performances.

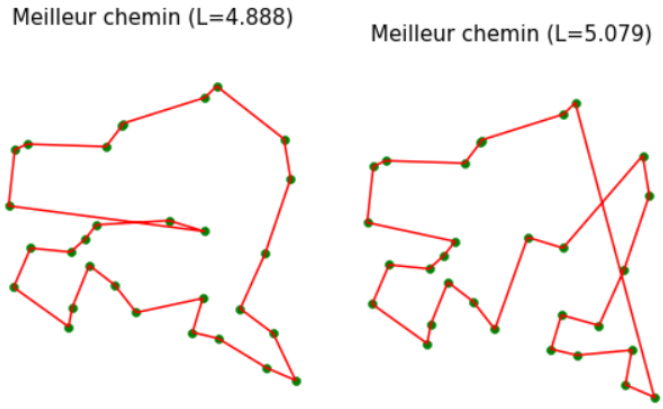

FIGURE 4.2 – Mutations par taux (gauche) et recuit simulé (droite)

4.2 Comparaison colonie de fourmis - algorithme génétique

L'algorithme génétique et celui de la colonie de fourmis s'exécutent en temps comparables, lorsqu'on utilise 10 fourmis par itérations pour 20 à 30 villes, ce qui est un nombre largement excessif (5 fourmis suffisent). Pour des petits nombres de villes, l'algorithme génétique arrive à des solutions plus élégantes.

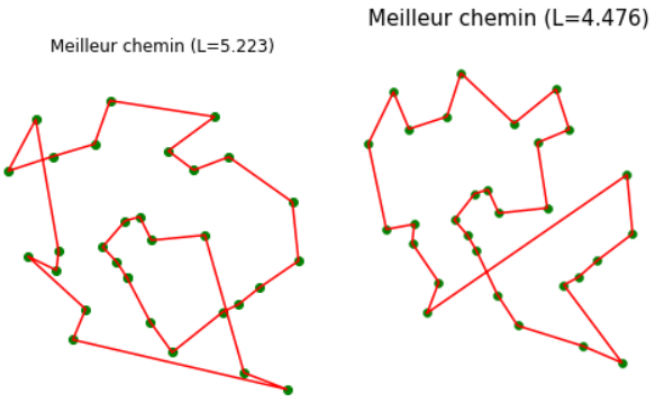

FIGURE 4.3 – Solution des fourmis (gauche) et de l'algorithme génétique (droite) pour 30 villes

En revanche, quand le nombre de villes est nettement plus grand, l'algorithme génétique peine à améliorer ses solutions et prend énormément de temps. L'algorithme de la colonie de fourmis, quant à lui, présente des solutions qui, même si elles nécéssitent quelques corrections simples à repérer, semblent abouties.

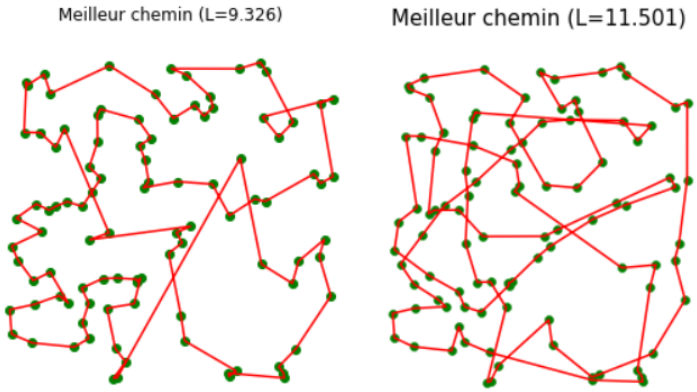

FIGURE 4.4 – Solution des fourmis (gauche) et de l'algorithme génétique (droite) pour 100 villes

Pour obtenir les graphes de la figure 4.4, l'algorithme de la colonie de fourmis a fait 10 000 itérations (11 min) contre 50 000 (36 min) pour l'algorithme génétique, donc pour donner une meilleure réponse, la colonie de fourmis est beaucoup moins gourmande.

5 Prolongements

Augmenter la dimension du problème

Une fois les méthodes approchées implémentées et comparées, nous avons vu que ces résolutions ne s'appliquaient qu'à un problème en deux dimensions (les villes ont 2 coordonnées). Nous avons donc voulu pouvoir résoudre ce problème en dimensions quelconques. Il s'agissait alors d'implémenter une seconde fonction longueur calculant la distance entre deux villes. La commande *longueur = longueurd* permet d'utiliser cette nouvelle fonction longueur et donc tester l'algorithme génétique en dimension quelconque. On peut alors résoudre le problème du voyageur de commerce en 3D comme le montre les figures ci-dessous.

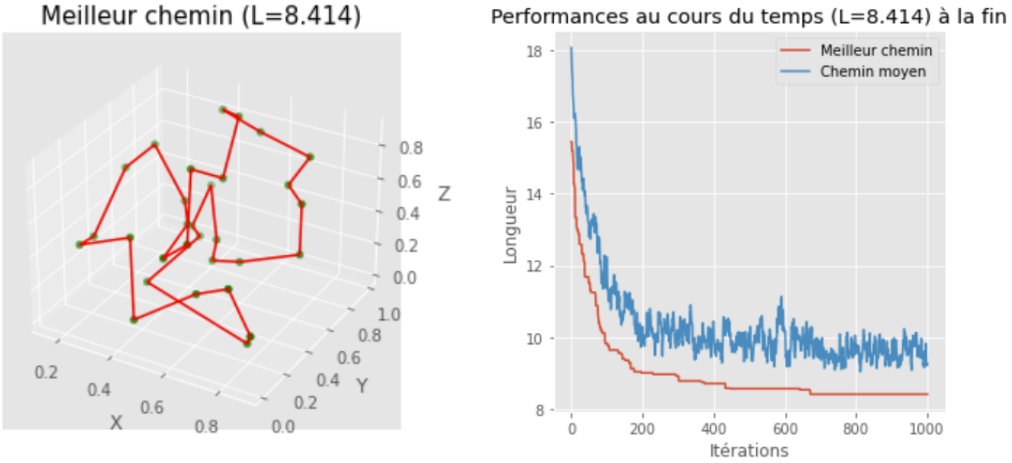

FIGURE 5.1 – Résultats en 3D

Réparer les erreurs évidentes

Une manière simple d'améliorer les solutions que nous proposent nos algorithmes, serait d'implémenter une fonction qui repère les endroits où un chemin se recoupe lui-même. En effet, en inversant alors l'ordre de deux points biens choisis, on peut « décroiser » le chemin, et ainsi se garantir un chemin plus court. Cette solution ne fonctionne évidemment qu'en deux dimensions, puis qu'il est extrèmement peu probable que deux segments se coupent dans à espace à 3 dimensions ou plus.