Projet IA: Implémentation du problème du Voyageur de Commerce via un algorithme génétique

Binôme: Corentin ROY, Pauline TEOULLE - M2 WIC

Méthode de travail

Pour mener à bien ce projet nous nous sommes appuyés sur de nombreux travaux et analyses existants. Nous avons choisi de nous renseigner individuellement puis de développer sur papier la logique théorique de notre algorithme à l'aide des connaissances récoltées.
En parallèle de l'implémentation de notre algorithme, nous avons été amenés à effectuer des modifications sur la logique théorique de ce dernier. Nous allons donc vous présenter dans la partie suivante le fonctionnement final de notre algorithme en omettant les étapes intermédiaires.

Fonctionnement de l'algorithme

Représentation du problème (encodage)

La première étape à faire était de trouver un moyen de représenter le problème. Nous avons choisi de partir sur un tableau de ville, une ville étant représentée par un nom et des coordonnées.

Cela nous a permis de visualiser notre problème plus aisément par la suite.
Chaque ville possède un numéro unique (id) nous permettant d'obtenir une représentation simple de cette dernière. C'est cet id qui représentera la ville tout au long du traitement.

Une fois le problème de la représentation des villes résolu, nous avons choisi un point de départ et nous avons représenté nos solutions par un tableau (dont l'ordre est important) de couple de valeurs (les valeurs sont les id des villes).
Le tableau suivant correspond donc à une suite de villes : [0, 5][5, 8][8, 1][1, 2][2, 4][4, 9][9, 6][6, 7][7, 3][3, 0].

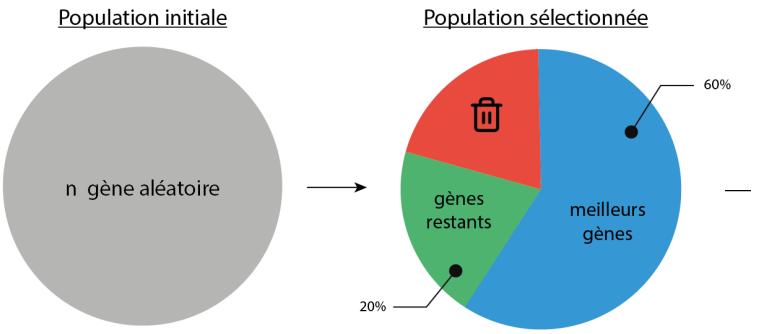
Ce tableau est la représentation de départ d'une solution. Nous générons donc X solutions aléatoires, qui correspondent à notre population de départ.

Sélection

Une fois notre population de départ générée, nous avons évalué l'aptitude de chacune de nos solutions et les avons ordonnées selon leurs poids (les meilleures solutions ont un poids très petit). Ici, la distance totale du chemin parcouru est le poids (sa valeur). A partir de cela, notre solution est représentée comme un tuple contenant sa représentation et son poids (représentation, poids).
(ex : ([0, 5, 8, 1, 2, 4, 9, 6, 7, 3], 209.27)

Ici, nous n'avons pas représenté le retour à la ville de départ pour faciliter notre croisement mais ce retour sera bien pris en compte quand nous en aurons besoin.

Nous avons donc notre population ordonnée selon leur poids et voulons en générer une nouvelle. Nous sélectionnons 60% de nos meilleures solutions (celles dont le poids est le plus petit), 20% de solutions moins bonnes choisies aléatoirement dans les 40% restants. Nous sélectionnons 80% de la population totale pour la prochaine étape, les 20% restants sont jetés et seront régénérés plus tard.

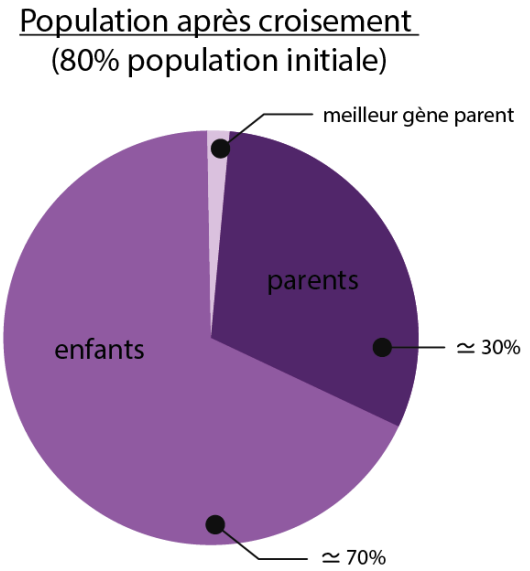


Nous avons choisi cette génération suite à de nombreux tests car nous avons observé que notre algorithme avait trop souvent tendance à être bloqué par un minimum local.
Pour résoudre ce problème, une des solutions est la régénération aléatoire d'une partie de la population. Une fois le croisement effectué nous allons régénéré les 20% restants de la population manquante.

Croisement

Une fois que nous avons sélectionné 80% de la population, nous effectuons un croisement.
Pour cela, nous affectons des probabilités de tirage pour chacune de nos solutions.
Nous utilisons une fonction de numpy sur une population ordonnée selon leur poids de solution afin d'obtenir des probabilités par rang et non par poids.
(ex : `np.geomspace(1,30,num=len(population))`).
Cette fonction utilise une échelle de valeur et une fonction logarithmique pour affecter des probabilités aux solutions. Cela permet d'éviter des solutions avec des probabilités trop éloignées ou au contraire trop proches comme cela aurait pu être le cas si nous avions basé nos probabilités sur le poids de la solution.

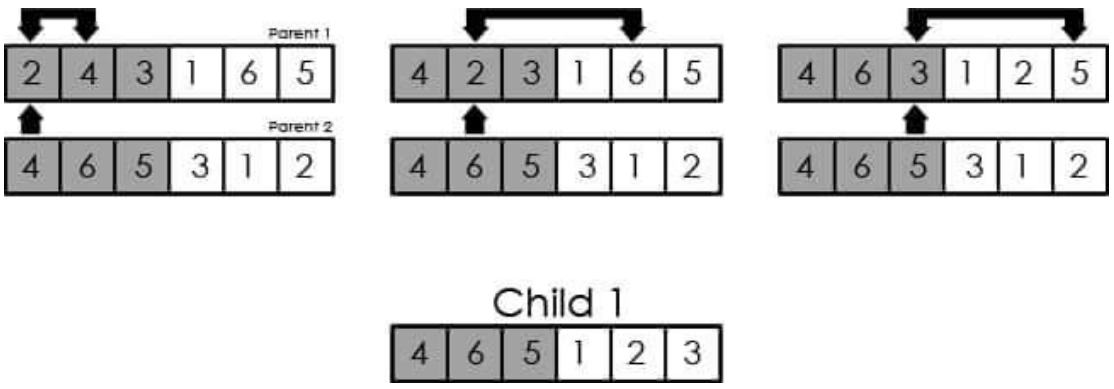
Nous itérons ensuite sur i= 70% de la population afin d'obtenir 70% de notre population par croisement. Nous effectuons un tirage avec remise en choisissant 2 parents en fonction de leurs probabilités. Nous avons fait en sorte que 2 parents génèrent 2 enfants et que 2 parents ne peuvent se reproduire qu'une seule fois ensemble par génération.



Une fois 70% de notre population croisée, nous sélectionnons la meilleure solution parent et la plaçons dans la génération suivante pour sauvegarder une tendance favorable par la suite.

Nous complétons ensuite notre population en choisissant environ 30% des parents restant selon leur probabilité affectée plus tôt. Cela permet de laisser une chance aux parents moins performants de subsister.

Notre opérateur de croisement marche avec un point de croisement aléatoire entre le premier et le troisième quart de la solution. Cela permet d'avoir la garantie qu'une partie suffisante des solutions parents se retrouve dans chaque enfant tout en garantissant un aspect aléatoire puisque le point de croisement n'est pas fixe. Une fois le point de croisement déterminé, nous échangeons la première partie du premier parent avec la première partie du deuxième parent selon le schéma suivant.



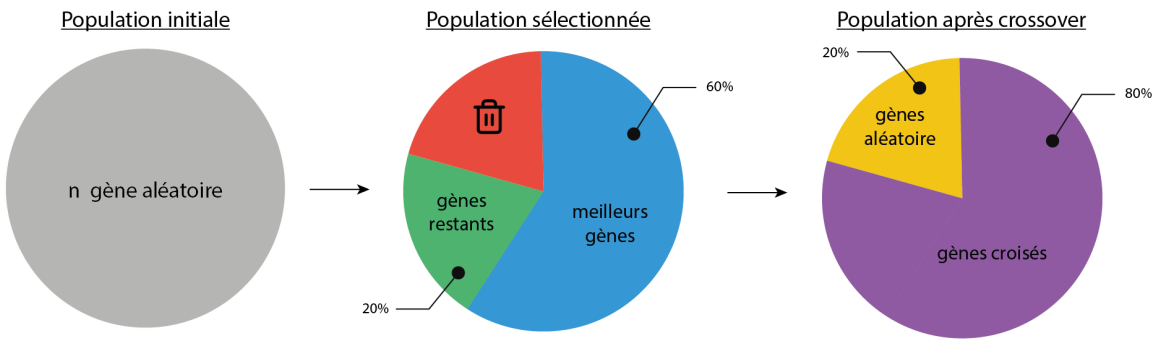
Pour chaque ville, on effectue un échange de la gauche vers la droite. Si la ville apportée existe ailleurs, cette dernière est remplacée par la ville qu'elle remplace.

Mutation

Une fois nos enfants obtenus, nous appliquons une probabilité de mutation afin de ne pas rester bloqué dans un minimum local.
S'il y a une mutation, deux villes sont choisies aléatoirement dans la solution enfant (sauf la ville de départ) et sont ensuite inversées.

Population finale

A cette étape là, nous avons reconstruit 80% de notre population initiale dont 70% sont issus de croisement et 30% des parents de la population précédente. Il nous manque donc 20% de population. Nous générons donc aléatoirement les 20% restants.

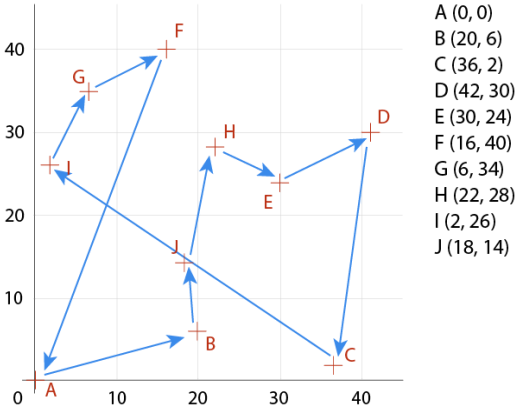


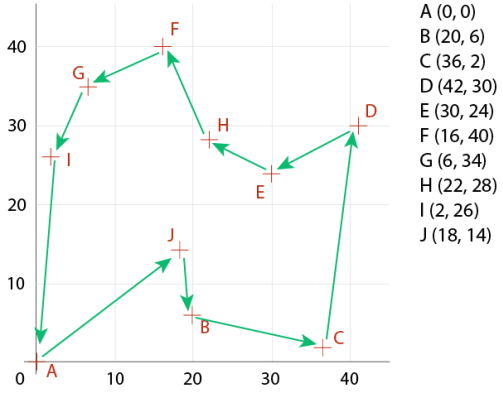
Notre population est désormais complète.
A chaque population générée, un enregistrement est effectué dans un fichier JSON. Cet enregistrement nous permettra ensuite de visualiser l'évolution des populations.
Nous réitérons les étapes précédentes jusqu'à un nombre d'itérations que nous avons fixé.

Analyse des résultats

Comparaison avec une algorithme glouton

Pour analyser les performances de notre algorithme, nous avons décidé de créer un problème unique sur lequel nous allons pouvoir effectuer nos tests en gardant un point de référence. Ce problème concerne 10 villes dont le point de départ est une ville A(0,0).
Une représentation est disponible ci-dessous.

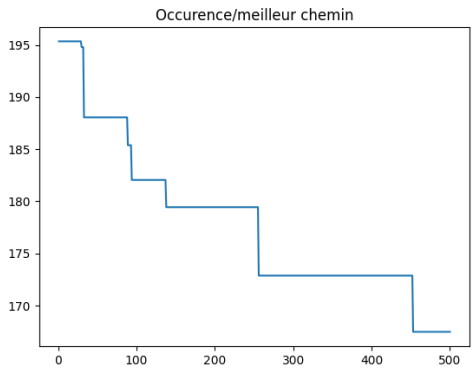


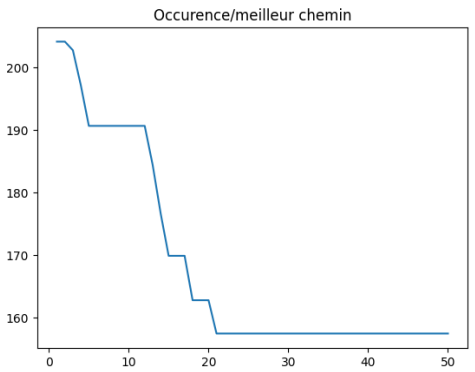


Vous voyez ici une comparaison de la solution trouvée par un algorithme glouton (en bleu à gauche) et de la solution trouvée par notre algorithme (en vert à droite). L'algorithme glouton trouve un chemin ayant une distance minimale de 198 (poids) alors que notre algorithme en trouve une de 158 (poids). Une solution dont le poids est minimal est le chemin optimal pour ce problème. Notre algorithme est donc meilleur qu'un algorithme glouton.

Comparaison avec un algorithme aléatoire

Nous avons également comparé la convergence de notre algorithme avec celle d'un algorithme aléatoire afin de mettre en avant l'efficacité de notre algorithme.
Si la convergence de notre algorithme avait ressemblé à la convergence de l'algorithme aléatoire, cela nous aurait indiqué que notre algorithme ne possédait aucune valeur ajoutée et était incorrect.

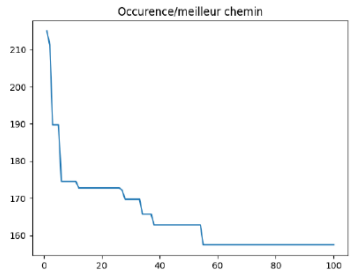




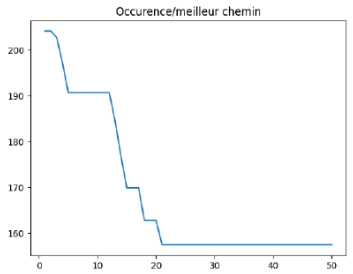
Le problème à résoudre est le même que précédemment. Nous avons utilisé une population de départ de 40 individus pour ce test.
Vous pouvez voir à gauche la courbe représentant l'évolution du meilleur chemin au fil des générations pour un algorithme aléatoire. On remarque que la courbe a une forme d'escalier et qu'il a besoin de beaucoup d'itération pour trouver un résultat loin d'être satisfaisant. L'algorithme génétique (à droite) converge beaucoup plus rapidement avant de stagner sur une solution plus efficace (qui est à ce jour la solution optimale connue de notre problème).

Influence de la taille de la population

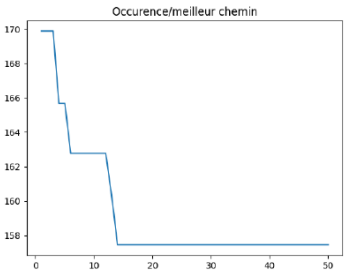
Nous avons lancé notre algorithme sur des tailles de population différentes afin de voir l'influence de la taille des générations. Pour 10 villes, il apparaît que la taille de population la plus efficace se situe entre 30 et 50 individus. En dessous, la convergence apparaît comme plus lente. L'algorithme met plus de temps à trouver une solution.
Au-delà de 50 individus, les gains obtenus par une taille de population plus grande sont effacés par le temps de calcul.
On ne note pas de différence suffisamment significative sur la solution trouvée ainsi que le nombre de générations pour y parvenir pour justifier l'emploi d'une grosse population.
Un des paramètres justifiant notre choix était également le temps d'exécution de l'algorithme, qui devait rester raisonnable.



Population : 25 individus



Population : 35 individus

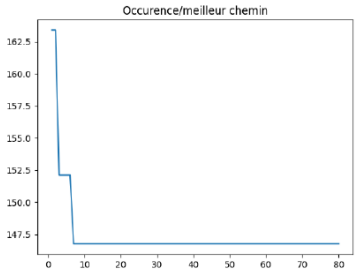


Population : 200 individus

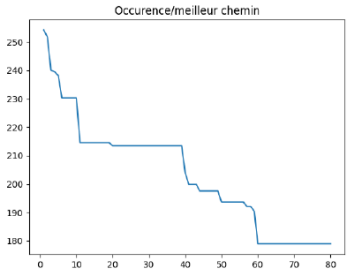
Vous pouvez voir que pour une population de 25 individus, l'algorithme commence à mettre beaucoup plus de temps pour trouver une solution. On remarque également qu'avec une population de 200 individus l'algorithme converge plus rapidement et trouve une solution 6 générations plus tôt que l'algorithme avec une population de 35 individus.
Cependant cette différence de génération, qui est minime, ne justifie pas à notre sens de presque tripler le temps de calcul qui passe de 3.87 secondes pour 35 individus à 10.23 secondes pour 200 individus sur 50 itérations.

Influence de la taille du problème (nombre de villes)

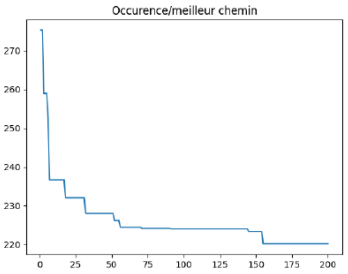
Nous avons également évalué l'influence de la taille du problème. Il semblerait que plus le problème est complexe, moins l'algorithme converge rapidement et subitement. Pour de petits problèmes, l'algorithme tombe très vite sur une solution et est peu souvent amené à en changer.
Plus le problème devient complexe, plus l'algorithme a tendance à stagner et à trouver de meilleures solutions. On remarque que lesdites solutions ne sont pas forcément bien meilleures. Il a tendance à progresser petit pas par petit pas comme sur le schéma ci-dessous dont la population initiale était de 40 individus.



Problème avec 8 villes



Problème avec 11 villes

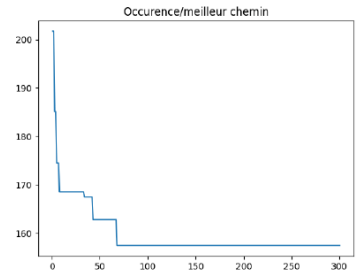


Problème avec 14 villes

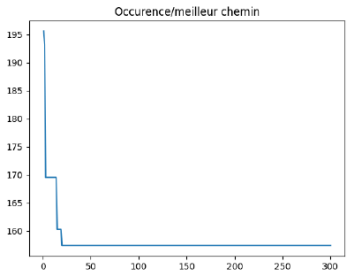
Influence du taux de mutation

L'évaluation de l'influence du taux de mutation a été quant à elle un peu plus difficile. De ce que nous avons pu lire, le taux de mutation des algorithmes génétiques se situe entre 0.5% et 1% pour calquer au mieux la réalité et ne pas trop soumettre l'algorithme au hasard. Cependant, nous avons observé de meilleurs résultats avec un taux de mutation situé entre 10% et 30%.
En regardant en profondeur les mutations effectuées, nous avons remarqué que dans certains cas, une solution plus efficace existait en inversant par exemple la deuxième ville et la dernière ville. Cette modification très précise était difficilement accessible par le biais des croisements puisqu'il suivait presque tous un pattern commun. Les seules solutions étaient qu'un parent régénéré possède cet attribut et soit sélectionné pour un croisement ou bien que la mutation effectue ce changement.

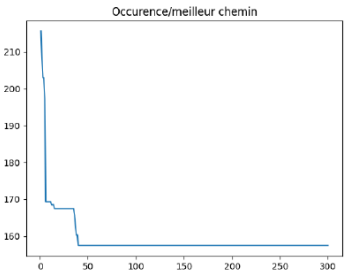
Nous avons observé que l'augmentation du taux de mutation permettait à l'algorithme de se sortir plus facilement du minimum local.
Cependant, un taux de mutation trop élevé réduit la convergence de l'algorithme. Nous avons vu que certaines fois, une solution intéressante était transformée à cause de cette mutation et ralentissait donc la convergence de l'algorithme. L'utilisation de l'élitisme semble pallier ce problème. En effet, si une solution est sélectionnée peu importe si une meilleure solution enfant est gâchée par la mutation, il existe une forte probabilité qu'il engendre un enfant aussi intéressant au fil des générations.



Taux de mutation : 1%



Taux de mutation : 15%



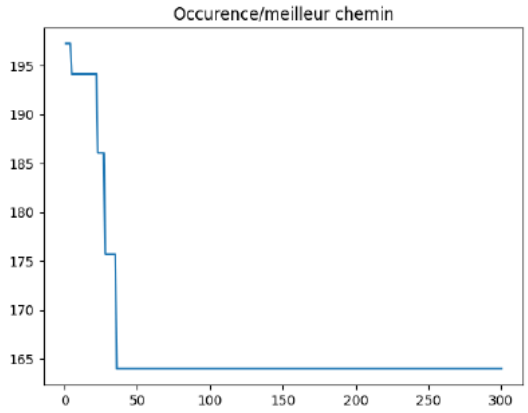
Taux de mutation : 50%

Influence du taux de croisement

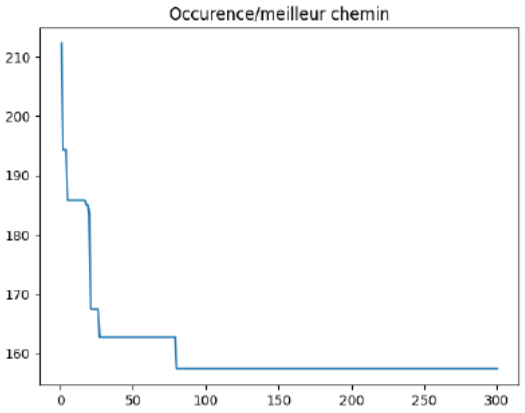
La dernière chose que nous avons cherché à savoir, c'est l'influence du taux de croisement. Actuellement 70% de la population est issue de croisements mais qu'en est-il si cette proportion venait à augmenter ou diminuer ?

Pour un taux de croisement trop faible, l'algorithme peine à trouver une bonne solution et à se sortir de minimums locaux. Cela réduit l'avancée vers une solution au fil des générations. L'influence de la mutation est aussi grandement réduite.

Un taux de croisement trop élevé ne conserve pas assez les bons éléments des générations précédentes. On conserve donc de moins bonnes solutions et on réduit par conséquent l'influence que pouvait avoir la génération précédente dans les générations futures.
L'algorithme met en général un peu plus de temps à trouver une solution intéressante mais y parvient tout de même.



Taux de croisement : 40%



Taux de croisement : 95%

Conclusion

Pour conclure, les algorithmes génétiques offrent une solution simple pour résoudre des problèmes complexes. Ce type d'algorithmes convient plutôt bien au TSP puisque la représentation du problème et son évaluation restent assez simples.
Cependant, il ne semble pas s'adapter à tous les problèmes. Il semble peu intéressant lorsque l'évaluation des individus prend en compte énormément d'éléments ou lorsqu'il faut engendrer des solutions multidimensionnelles. Cela rendrait l'implémentation d'un tel algorithme beaucoup trop complexe et augmenterait considérablement le temps de calcul.

D'autre part, pour en revenir à notre analyse, nous prônons le pragmatisme et la mesure en soulignant le fait que notre analyse repose sur des tendances que nous avons pu observer. Ce que nous avons évalué est le fruit d'expériences non protocolaires et il est probable que certaines de nos analyses soient biaisées par différents paramètres : un taux d'itération des expériences faibles (≈10 par expérience), la résolution de problèmes trop simplex et/ou peu variés, ou encore l'intervention humaine à chaque étape des expériences.