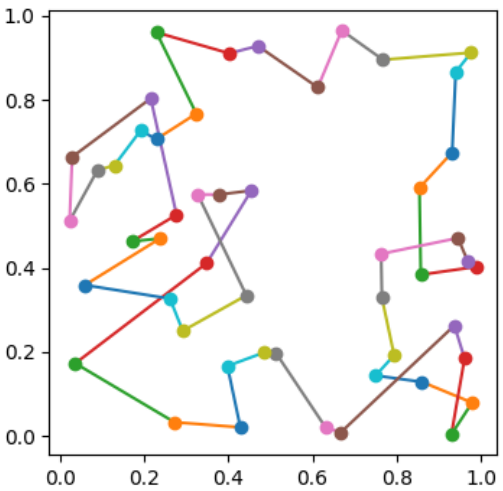
Blanchard Allan
Raad Josué

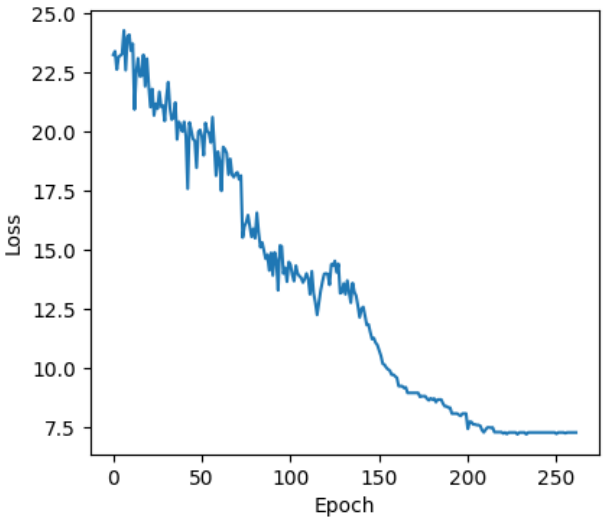
Faculté des Sciences & Techniques

master ISICG limoges

Rapport de projet IA2

Algorithme génétique pour le problème du voyageur de commerce





Université de Limoges, FST, Master 1 Informatique – Année 2021/2022

Encadrant : M. Karim Tamine

Sommaire

Introduction

Ce projet réalisé dans le cadre de l'UE IA2 consiste à créer un algorithme génétique afin de proposer une solution au problème du voyageur de commerce.

Le voyageur de commerce

Si l'on considère un graph non orienté complet de N sommets, alors il faut trouver le circuit hamiltonien le plus court.

Autrement dit, les sommets sont nos villes, et le voyageur doit effectuer un trajet passant une et une seule fois par toutes les villes et regagner son point de départ à l'étape finale.

La somme des distances entre les villes nous donne le coût du parcours.

Trouver le parcours qui est la meilleure solution à ce problème requiert de tester toutes les combinaisons de chemins possibles, cela provoque une explosion combinatoire et il n'est possible de calculer la solution que pour des graphs simples. Ce problème est d'ailleurs NP-Complet.

L'algorithme génétique

La méthode que nous utilisons est un algorithme génétique. Ce type d'algorithme inspiré par le processus biologique de la sélection naturelle peuvent proposer des solutions à un problème d'optimisation.

Utiliser un tel algorithme ne garantie pas d'obtenir la meilleure solution au problème, mais permet en revanche d'obtenir une solution proche de l'optimal en un temps raisonnable (si les hyperparamètres sont correctement configurés)

Structure du projet

Scripts python

Le projet contient dans le dossier `./src` tous les scripts python utilisés

- `data_manager.py` : Utilisé pour générer des données (les villes). On peut générer une carte aléatoirement, un arrangement particulier ou lire depuis un fichier
- `view_manager.py` : Contient le code relatif à la visualisation de l'avancée de l'algorithme. Il permet d'afficher des graphiques en temps réel.
- `genetics.py` : Les composantes relatives à la gestion du génome des individus, mutation, croisements…
- `individual.py` : la classe individu, permet de calculer le coût de parcours d'un individu pour un graph donné
- `main.py` : le programme principal, initialisation, lancement de l'apprentissage, affichage des résultats

Exécution du code

Le programme requiert les modules python suivants :

- numpy (gestion des tableaux)
- tqdm (barres de progression)
- matplotlib (affichage graphique des données)

Pour lancer un apprentissage, on peut exécuter la commande suivante dans un terminal à la racine du projet :

```
python3 ./src/main.py
```

Structures de données

Graph

Les sommets du graph sont enregistrés sous la forme d'un tableau de coordonnées 2D en valeur flottante. Ces valeurs sont présentes dans la classe `dataManager`. Il n'y a pas de conditions sur les bornes des coordonnées ou la taille du graph.

Individus

Un individu contient un génome qui est présenté dans le code sous la forme d'un tableau d'entiers uniques. La position des entiers dans le tableau donne l'ordre de parcours et la valeur des entiers donne le numéro du sommet parcouru dans le graph.

Par exemple pour le graph `[(0,0) (1,1) (5,5)]` et l'individu `[1,2,0]` : cela signifie que l'on parcourt d'abord le sommet en `(1,1)`, puis le sommet en `(5,5)`, ensuite le sommet en `(0,0)` et enfin, on revient implicitement au sommet en`(1,1)`.

Une valeur ne peut être présente qu'une seule fois par individu, sinon cela signifierait que l'on passe plusieurs fois par la même ville.

Étant donné que les individus ne sont qu'un conteneur pour les génomes, nous nous référons aux deux de manière interchangeable.

Évaluation

L'évaluation de la distance parcourue se fait au moment de la création de l'individu. On utilise la somme des distances euclidiennes entre les villes dans l'ordre de parcours définis par l'individu. Étant donné que l'on cherche à minimiser cette distance, on l'appelle également perte (ou loss) ici.

Principe de l'algorithme

L'algorithme génétique recherche une solution optimale à un problème en brassant des populations d'individus et en sélectionnant les meilleurs génomes pour la génération suivante. Ce mécanisme se répète jusqu'au critère d'arrêt.

La première population est générée aléatoirement. Parmi les individus, certains ont une perte plus faible que la moyenne. Alors certains d'entre eux sont sélectionnés et utilisés pour produire une nouvelle population d'individus.

Ce processus est permis par les méthodes de croisement (ou crossover). On mélange les gènes de deux individus que l'on appelle les parents. Un ou plusieurs enfants sont produits et leur génome est un mélange de celui des parents.

Pour empêcher une trop grande uniformité génétique, des mutations sont introduites dans les enfants en changeant aléatoirement certains gènes.

Par ce principe, on espère ainsi garder uniquement les gènes qui sont les plus intéressants en reproduisant les meilleurs et en se débarrassant des individus médiocres génération après génération.

Il y a deux facteurs qui peuvent principalement affecter la vitesse de la résolution du problème :

- Les méthodes de sélection, mutation, croisement.
- Et les valeurs des hyperparamètres

Nous discutons plus en détail de leur influence et des valeurs optimales dans la section dédiée.

Programme principal

La boucle principale charge les données, lit les paramètres et lance l'évolution génétique avec un affichage en temps réel.

À chaque époque, le meilleur individu est enregistré s'il est meilleur que celui de la génération précédente. Ainsi, si le brassage génétique produit un moins bon résultat, on garde tout de même le meilleur chemin.

Pour l'affichage, c'est le meilleur individu de chaque génération qui est utilisé pour plotter la loss et afficher le graph. Ainsi, on peut constater que l'algorithme régresse parfois pendant l'évolution.

Au début du programme, on peut régler les données utilisées pour le graph, les conditions d'arrêt, les conditions d'affichage et les paramètres génétiques.

Ce sont les fonctions `run_genetic()` et `newGen()` qui sont responsables de l'évolution à proprement parler, les populations sont modifiées époque après époques.

On peut définir comme condition d'arrêt une époque maximale, ou alors une valeur atteinte que l'on considère suffisante.

Par exemple, avec un cercle de diamètre 1, le critère d'arrêt optimal est une distance de 3,14. Mais avec un graph aléatoire, le paramètre optimal ne peut pas être déterminé à l'avance.

Analyse

Analyse des choix de méthodes

L'évolution génétique se fait en trois temps : Sélection, croisement, puis mutation. Dans la section qui suit, nous analysons les méthodes employées et justifions nos choix. Lorsque plusieurs méthodes ont été implémentées, nous détaillons les avantages et inconvénients de chacune.

Sélection

Pour sélectionner les meilleurs parents afin de produire la génération suivante, on garde un pourcentage fixe des meilleurs individus. Ensuite, en fonction du nombre d'individus gardés et du nombre d'enfants souhaités, on attribue à chaque parent un partenaire parmi les parents présents.

Pour indiquer les gènes des individus à croiser, on utilise une matrice : chaque ligne correspond à un parent et les valeurs contenues dans cette ligne référencent un autre individu parmi ceux sélectionnés.

Les valeurs sont remplies avec d'autres parents choisis au hasard parmi la sélection. Ainsi, le meilleur individu (ligne 0) se reproduit avec un sous groupe aléatoire des autres individus sélectionnés. Puis le second meilleur (ligne 1) se reproduit… Ainsi jusqu'à ce que la taille de population maximale soit atteinte.

On s'assure qu'un parent ne peut pas avoir de croisement avec lui-même. Dans le cas où le nombre de combinaisons entre parents n'est pas assez grand pour produire la génération suivante, on remplit les espaces manquants avec d'autres parents sélectionnés au hasard parmi les meilleurs.

Et dans le cas où il y a trop de combinaisons de parents pour la génération suivante, seuls les meilleurs parents ont le droit de se reproduire, mais les partenaires sont choisis au hasard parmi les autres meilleurs.

Ainsi, on laisse toujours la priorité de reproduction aux individus les mieux classés.

Les croisements sont faits de telle sorte que deux parents `A` et `B` puissent se reproduire deux fois. En effet, les combinaisons `cross(A,B)` et `cross(B,A)` ne produisent pas les mêmes enfants. C'est pour brasser le plus possible les gènes dans la génération suivante tout en préservant au mieux les caractéristiques de chaque parent.

Croisements

Nous avons implémenté trois algorithmes de croisements afin de simuler les recombinaisons génétiques. L'enjeu de tels algorithmes est de conserver au maximum l'ordre des gènes des deux parents.

En effet, c'est bien ici l'ordre dans lequel apparaissent les gènes qui dicte la performance de l'individu. Et étant donné qu'un indice de gène ne peut apparaître qu'une seule fois dans le génome, il faut à tout prix minimiser les collisions.

Par exemple : Si l'individu `A=(1,2,3,4)` se croise avec l'individu `B=(4,3,2,1)`. Ensuite, une manière naïve de procéder serait de prendre la première moitié du génome de A et de la combiner avec la seconde moitié du génome de B pour donner l'enfant `C=(1,2,2,1)`.

Seulement voilà, `C` ne satisfait pas les critères de parcours du graph : il passe plusieurs fois par les mêmes sommets, et ne parcourt pas la totalité des sommets. Une rectification des gènes de C est nécessaire pour qu'il soit un individu bien formé [1].

Après modification (potentiellement aléatoire) des gènes de `C` on obtient par exemple `C'=(1,2,4,3)`. Le nouvel individu est viable, mais il perd la plupart des caractéristiques du parent `B`.

Implémentation des croisements

Comme nous venons de le voir, limiter les collisions dans les croisements de parents est crucial. Nous allons analyser les solutions que nous proposons :

- Médiocre - Fonction `crossover3()` :

Dans cette méthode, on utilise une approche similaire à l'exemple naïf ci-dessus. Un point de découpe 'cut' est choisi aléatoirement et dans l'intervalle [0..cut] les gènes du parent A sont présents. Puis dans l'intervalle [cut..N] les gènes du parent B sont présents.

[1] C'est cruel quand on y pense.

Si le génome de A est bien préservé dans la première moitié, celui du parent B est souvent totalement changé, cela provoque la perte de beaucoup d'information génétique dans la seconde moitié du génome.

Cependant, ce croisement n'étant pas symétrique, les informations génétiques issues du parent A sont en moyenne tout autant préservée que celles issues du parent B [2]. Mais les deux enfants C_{AB} et C_{BA} perdent tous deux de l'information.

En somme, cette méthode satisfait les critères minimum pour faire tourner l'algorithme, mais s'avère mauvaise pour trouver une solution rapidement.

- Passable - Fonction `crossover2()` :

Ici le principe se révèle semblable à la méthode `crossover3`, mais on tire au hasard deux points de coupe : `c1` et `c2`. Comme le génome est divisé en morceaux de plus petite taille, cela réduit les probabilités de collisions. Les gènes de `A` sont gardés dans l'intervalle `[0..c1] et [c2..N]`, tandis que ceux de `B` sont conservés dans l'intervalle `[c1..c2]`.

Cette méthode souffre des mêmes problèmes que la précédente dans une moindre mesure.

- Meilleur - Fonction `Crossover()` :

Cette dernière méthode utilise l'aspect cyclique des gènes pour minimiser les collisions.

Étant donné que les gènes sont indexés sur des indices de 1 à N, possèdent des valeurs entre 1 et N, et sont uniques dans chaque génome, alors on peut dégager des cycles entre A et B.

Le principe est le suivant :

On crée des cycles selon le schéma suivant
On a un élément de A, on regarde la valeur dans la même position en B.
On se déplace jusqu'à la position de A contenant la même valeur.
On ajoute cette valeur au cycle actuel
Lorsqu'on revient à l'élément original de A, cela signifie que l'on a bouclé un cycle.
S'il reste des éléments qui ne sont pas encore dans un cycle, on recommence.
Pour chaque cycle, ajouter alternativement les valeurs du cycle de A, puis les valeurs du cycle de B.

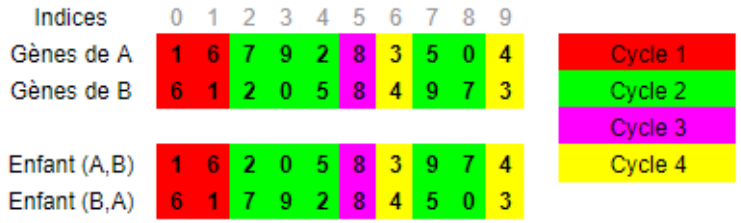


Figure 1 - Tableau explicatif du croisement à base de cycles

Le principal avantage de cette méthode de croisement est que les collisions sont totalement évitées. Toutes les informations génétiques contenues dans la position des gènes sont sauvegardées.

De plus, cette méthode permet de garder au même endroit les valeurs intéressantes. Sur la figure ci-dessus, on constate que la valeur 8 est restée inchangée entre les parents et les enfants.

Si la même valeur revient deux fois au même endroit sur deux parents compétitifs, on veut la garder, car ce gène représente un avantage génétique.

Le désavantage de cette méthode est la performance légèrement amoindrie (environ 15 % selon nos tests), parce qu'il faut trouver les cycles avant de commencer une recombinaison de gènes.

Mutations

Les fonctions de mutations sont appelées pour chaque enfant produit après les croisements. Tout comme pour les croisements, nous avons créé plusieurs méthodes de mutation.

Tous les tirages aléatoires sont comparés au paramètre `mutation_rate` dans le programme principal. Le tirage est considéré comme un succès si `rand()<mutation_rate`.

[2] Car `cross(A,B) != cross(B,A)`

Le but de la mutation est d'introduire de la diversité génétique au fil du temps, pour éviter que l'algorithme ne stagne en brassant toujours des individus aux génomes similaires. Une bonne fonction de mutation doit modifier uniformément les gènes dans la population [3] .

- `Mutation3()` :

Cette méthode de mutation fait un premier tirage aléatoire pour savoir si une mutation va se produire ou non. Si le tirage est un succès, alors un nombre aléatoire de gènes sont sélectionnés et sont intervertis deux à deux. Expérimentalement, cette méthode est peu efficace et nécessite des plus grandes valeurs du taux de mutations.

- `Mutation2()` :

Lorsque le tirage aléatoire est un succès, deux gènes sont pris au hasard et sont intervertis. Cette méthode est expérimentalement légèrement meilleure que la précédente et nécessite aussi un taux de mutation plus élevé pour avoir un effet significatif.

- `Mutation()` :

Un tirage est effectué pour chaque gène, si le tirage est un succès alors ce gène est interverti avec un autre choisit aléatoirement. Cette méthode est très sensible au taux de mutation et permet de modifier les gènes de manière plus équitable entre les individus. C'est expérimentalement celle qui marche le mieux.

Analyse des choix de paramètres

Conditions expérimentales

Sauf si précisé autrement, nous avons utilisé les méthodes `mutation()` et `crossover()`, car ce sont celles qui produisent les meilleurs résultats.

Pour comparer les performances, nous utilisons un graph prédéfini : un cercle de N points et de diamètre 1. Le trajet optimal sur ce cercle a une distance inférieure à 3.14. La figure 1 représente une illustration d'un tel graph.

Les conditions d'arrêt sont 1000 époques, ou alors une distance légèrement supérieure au chemin optimal connu (5).

Les paramètres utilisés par défaut sont :

+variable	Val par défaut	Description
data_size	50	La taille du graph
max_epoch	1000	Le nombre max d'époques avant que l'évolution ne s'arrête
target_loss	5	La distance souhaitée avant que l'évolution ne s'arrête
keep_percent	0.18	Le pourcentage d'individus à garder pour les croisements de la prochaine génération
population_size	100	La taille de la population initiale
mutation_rate	0.006	Le taux de mutation

Tous les tests ont été répétés plusieurs fois pour vérifier que les résultats discutés restent cohérents malgré le caractère aléatoire de l'algorithme.

Taille de la population

La taille de la population influence directement la vitesse de convergence et la qualité de la solution. Des populations trop petites par rapport à la taille du graph ne disposent pas d'une diversité génétique suffisante pour converger aisément. Cependant, il y a un point à partir duquel augmenter la taille de la population n'a que peu ou pas d'effet sur la qualité de la solution, ni sur la vitesse de convergence.

On voit sur la figure 2 qu'à partir de 30 individus, une solution intéressante est systématiquement trouvée. Tandis qu'à partir de 80 individus, la vitesse d'obtention de cette solution ne croît plus.

Choisir un nombre d'individus faible permet de diminuer le temps de calcul. Ça ne sert à rien de faire évoluer 500 individus si le résultat est le même qu'avec 100.

De plus, on remarque que sur des populations très importantes, le nombre d'époques pour obtenir la solution croît légèrement (en plus de prendre plus de temps à calculer chaque époque).

[3] Autrement dit, il faut éviter de bouleverser les gènes de seulement quelques individus, mais plutôt effectuer des petites modifications sur les gènes de tous les individus.

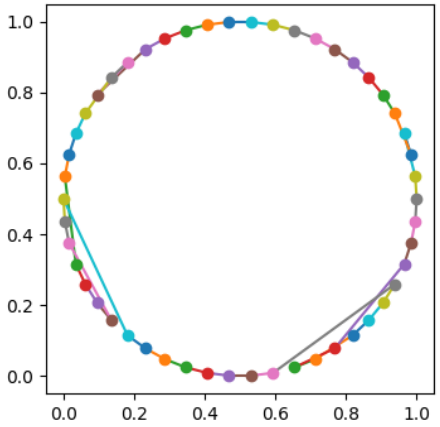


Figure 2 - graph arrangé en forme de cercle de distance optimale connue ~= 3.14

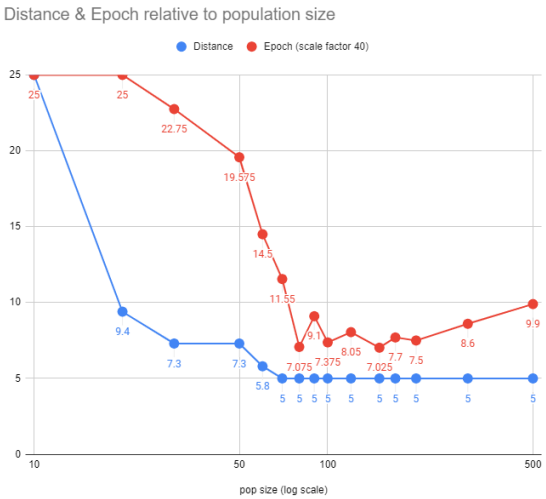


Figure 3 - Distance et nombre d'époques en fonction du nombre d'individus

Taux de mutation

Le taux de mutation est un paramètre très sensible. S'il est trop bas, il n'y aura jamais de mutation intéressante et l'algorithme va stagner à une solution peu optimale. S'il est trop haut, il ne va pas y avoir de convergence vers une solution, car les populations seront trop changées d'une génération à l'autre.

On constate sur la figure 3 qu'il y a vraisemblablement un taux de mutation idéal entre 0.002 et 0.009. Les valeurs en dessous de cet intervalle convergent certes, mais avec un nombre d'époques élevé. Tandis que les valeurs au-delà n'arrivent pas à converger malgré le grand nombre d'époques qu'elles utilisent. Taux de Croisements

Le taux de croisement est représenté par la variable `keep_percent`. À chaque époque, une partie de la population est sélectionnée pour les croisements qui mèneront à la génération suivante.

On constate sur la figure 4 que la qualité de la solution et la vitesse d'obtention dépendent tous deux directement du taux de croisement.

Une valeur aux alentours de 17 % semble optimale avec les paramètres actuels. Si la valeur est trop petite ou trop grande, alors on n'obtient pas de solution en temps raisonnable, ou pas de convergence.

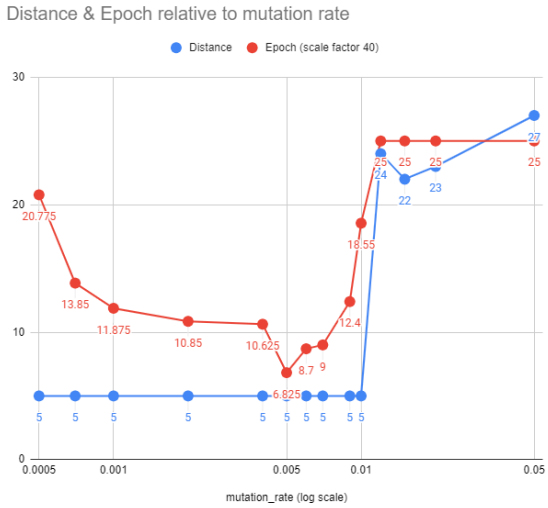


Figure 4 - Distance et nombre d'époques en fonction du taux de mutation

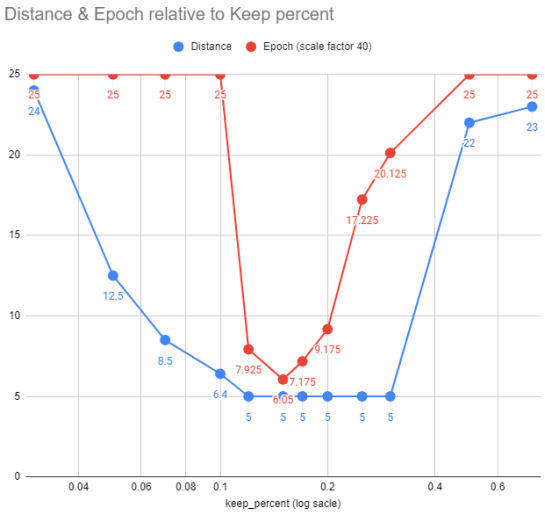


Figure 5 - Distance et nombre d'époques en fonction du taux de croisement