De Thierry RALAINARIVO] Master 1 _ n° 17807592

Contents

Rapport: Voyageur de commerce avec l'algorithme génétique

TP encadré par B. Daachi

May 2021

1 Introduction

Le problème du voyageur de commerce consiste à passer par un ensemble de villes en minimisant la distance totale du trajet. C'est un problème dit NP-complet, ce qui signifie qu'il n'existe pas d'algorithme en temps polynomial permettant de trouver une solution exacte à ce problème.
Pour essayer de résoudre ce problème, nous allons utiliser la notion de classes en Python. Les objets que nous allons manipuler sont les villes et les circuits.

2 Classe "ville":

Nous allons créer une classe Ville. Les villes sont définies par leurs coordonnées GPS (longitude et latitude), et leur nom. Puis en déduire la distance entre deux points avec leurs coordonnées.
J'ai remarqué qu'on est obligé de passer par les coordonnées géographiques pour pouvoir placer les villes sur un graphe. Les informations de départ et d'arrivée de l'énoncé ainsi que les distances sont insuffisants pour résoudre le problème. J'ai donc récupérer les coordonnées géographiques des villes qui nous intéressent.

3 Classe "gestionnaire du circuit":

Par la suite nous définissons une classe qui nous permettra de gérer les circuits. Un circuit est caractérisé par un ensemble de villes stocké dans la liste *villesDestination*. Il est important de mentionner que vu la nature de notre problème, l'ordre des villes a une importance ! Toutefois, un circuit étant une **boucle**, les circuits *Lille-Amiens-Rouen-Paris-Lille* et *Paris-Lille-Amiens-Rouen-Paris* sont **équivalents.**
Vous pouvez remarquer que notre circuit contient un attribut *fitness* et une méthode *getFitness*. C'est un attribut indiquant **la qualité d'un circuit**. Dans notre cas, **un circuit aura une meilleure fitness lorsque la distance totale sera faible**. En effet, l'objectif de notre algorithme est de trouver un circuit minimisant la distance totale de voyage. C'est la raison pour laquelle notre fonction de fitness retourne l'inverse de la distance totale. Ainsi, lorsque la distance diminue, alors la fitness augmente.
Une autre méthode qui mérite quelques précisions est la méthode *genererIndividu*. Comme nous l'avons expliqué dans la partie précédente, une des étapes principales de notre algorithme est la génération d'une population. Dans notre méthode nous remplissons une liste avec l'ensemble de nos villes puis nous les réordonnons de manière aléatoire en utilisant la fonction random.shuffle.
Nous continuons notre programme par la création d'une classe *Population*. Dans le cas général, une population est un ensemble d'individus. Dans notre cas cela correspond à un ensemble de circuits.

4 Classe "population":

C'est l'ensemble des circuits. La méthode *getFittest* nous retourne le circuit ayant **la plus grande fitness**, ce qui équivaut à la plus faible distance.

5 Classe "algorithme génétique ":

C'est la méthode d'optimisation qu'on a choisi.
Celle-ci contient les principales étapes d'un algorithme génétique :

1. L'évolution de la population
2. Le crossover
3. Les mutations
4. La sélection des individus à reproduire.

Tout d'abord l'attribut *tauxMutation*. Comme son nom l'indique, c'est la probabilité qu'une ville d'un circuit subisse une mutation. Dans notre cas, cela correspond à l'inversion de la position de deux villes dans le circuit. Le taux est assez faible car la probabilité d'obtenir une distance plus faible en inversant deux villes est peu élevée.
L'attribut *tailleTournoi* correspond à la taille des poules de notre tournoi. Le **tournoi** est une des méthodes possibles pour sélectionner les individus que nous souhaitons faire se reproduire. Il en existe d'autres telles que la sélection par roulette ou bien la sélection par rang.
La sélection par tournoi (que nous utilisons) fait affronter plusieurs individus sélectionnés au hasard. Dans notre cas nous organisons des tournois de 5 circuits et nous gardons le circuit avec la distance la plus faible (la fitness la plus élevée). Ainsi, dans le cas où nous faisons s'affronter 5 mauvais individus, cela laisse tout de même une chance à un de ces (mauvais) individus de pouvoir se reproduire.
Dernier attribut, l'*elitisme*. Dans un algorithme génétique l'élitisme correspond au fait de vouloir **conserver les meilleurs individus d'une génération à une autre**, afin d'être sûr de ne pas les perdre. Cela a pour avantage d'accélérer la convergence de l'algorithme au détriment de la diversité des individus.

Dans notre cas nous choisissons un 2-points crossover mais contrairement aux exemples donnés précédemment nous avons une contrainte supplémentaire : la cohérence de notre solution. En effet, l'individu issu de la reproduction doit avoir un circuit contenant toutes les villes possibles. Pour pallier à ce problème nous procédons de la manière suivante :

1. On choisit deux indices;
2. On recopie les villes présentes entre ces deux indices dans notre futur individu;

3. On complète les emplacements vides de notre nouvel individu par les villes manquantes.

6 Résolution du problème:

Pour résoudre le problème, voici les étapes effectuées:

1. on crée nos villes avec les coordonnées géographiques de chacune pour pouvoir les placer dans un repère orthonormé. Puis on pointe le gestionnaire de circuit sur chaque ville introduite.

2. on initialise la population à 100 circuits.
   On note que plus le nombre de circuit d'initialisation est grand, plus le résultat est *stable*, c'est-à-dire que les valeurs ne changent pas à chaque implémentation et appel du random_shuffle.

3. on fait évoluer notre population sur 100 générations.
   On peut ainsi déjà observer une amélioration au niveau des distances. En effet, la distance finale qu'on récupère ici est optimisée par rapport à la distance initiale.

4. on place les villes sur un plan selon leurs longitudes et leurs latitudes pour pouvoir visualiser la solution grâce à matplotlib.

5. on fixe la ville de départ: 'Lille'.
   Soit on met 'Lille' en première position comme sur le schéma, soit on implémente directement le programme. Comme le circuit fait une boucle fermée, on peut partir de n'importe quelle ville pour y revenir toute en respectant l'optimisation de la distance parcourue. Et dans ce cas, on choisi 'Lille' comme ville de départ et d'arrivée.

6. on implémente le programme.
   On note que le résultat n'est pas obligatoire le même entre deux implémentations car c'est un programme qui dépend beaucoup de l'aléa (random) qu'on invoque à chaque fois. Par contre, on peut affirmer que l'optimisation est assurée.

7. Résolution graphique :

jupyter voyageur_de_commerce_GA Dernière Sauvegarde : il y a une minute (modifié)

File Edit View Insert Cell Kernel Widgets Help Fiable

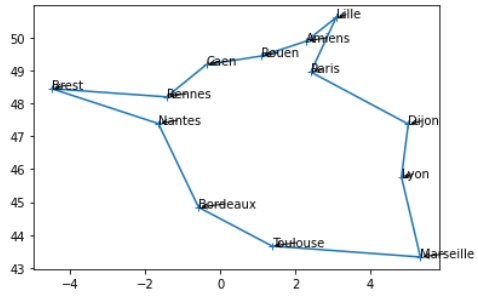
▶ Exécuter Code

```
    noms.append(noms[0])
    noms[0]='Lille' #par hypothèse, on commence par la ville de 'Lille'
    distance.append(distance[0])
    plt.plot(lons,lats,'+-')
print('\n')
print('En faisant l\'hypothèse que nous nous déplacions en ligne droite entre les villes, le circuit optimal est: \n')
for i in range(0, len(distance)):
    print(f'{noms[i]}')
plt.show()
```

```
Distance initiale : 4436.21718910335 km
Distance finale (optimisée): 2628.8891960789006 km


En faisant l'hypothèse que nous nous déplacions en ligne droite entre les villes, le circuit optimal est:

Lille
Paris
Dijon
Lyon
Marseille
Toulouse
Bordeaux
Nantes
Brest
Rennes
Caen
Rouen
Amiens
Lille
```



Ce document est édité sur LaTeX par Thierry RALAINARIVO