Université de Montpellier / Faculté Des Sciences

Algorithme génétique

Algorithmes d'exploration — HMIN 233

Suro François

8 décembre 2020

Dans ce TP nous allons appliquer l'algorithme génétique sur le problème du voyageur de commerce (TSP - travelling salesman problem)

1 Le problème du voyageur de commerce

Nous allons poser le problème de la manière suivante :

Étant donné N villes formant un graphe complet orienté et valué, et une ville de départ (un sommet de ce graphe), trouvez le chemin de coût minimum passant exactement une fois par toutes les villes avant de revenir à la ville de départ.

Quelques remarques :

— Le graphe est complet : on peut atteindre n'importe quelle ville à partir de n'importe quelle autre.
— Le graphe est orienté : le trajet de la ville A vers la ville B n'est pas le même que celui de la ville B vers la ville A.
— Le graphe est valué : la valeur de l'arc représente le coût.
— Comme la ville de départ/arrivé est fixé, on peut représenter le chemin avec N-1 éléments (dans quelle ordre je visite toutes les autres villes).

Dans le package `tsp` vous trouverez 2 classes à compléter.

`TSPMain` contient le main du programme. Pour vous faciliter la tâche, une représentation du graphe vous est fourni, ainsi que des fonctions pour générer, afficher et résoudre le problème avec un algorithme de brute force (force brute : on essaye toutes les possibilités). Le tableau *distances* représente le graphe, la première dimension est la ville de départ, la seconde dimension la ville d'arrivée et la valeur est le coût associé au déplacement.

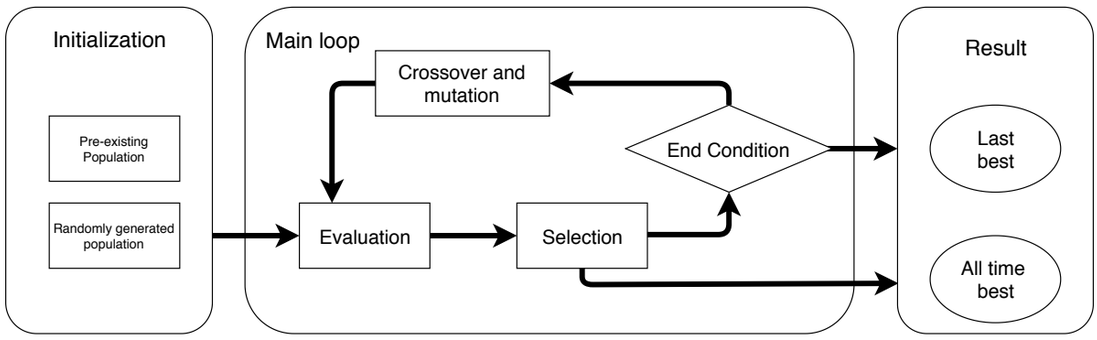
Vous appellerez les fonctions de votre classe algorithme génétique dans le main, et vous pourrez comparer ses résultats avec la fonction de brute force (sur un petit nombre de villes en tout cas ...)

`AlgoGenTSP` contiendra votre implémentation de l'algorithme génétique. Le constructeur prend en paramètres les données minimales du problème (le graphe sous forme de tableau et le nombre de villes), vous pouvez le modifier comme bon vous semble. Pour ceux qui ne sont pas habitués à Java, je vous propose une classe Individu qui vous servira à représenter les individus de votre population, cette classe implémente l'interface *Comparable* qui permet ensuite d'utiliser les algorithmes de tri de Java. Il vous suffira de remplir la fonction *compareTo* pour indiquer sur quel critères vous comparez les individus. L'utilisation de cette classe est complètement optionnelle.

Pour simplifier l'implémentation on considère que la ville de départ est toujours le sommet 0.

2 L'algorithme génétique

1. Écrivez le constructeur de la classe `AlgoGenTSP` (facile)
2. Écrivez une fonction qui crée un nouvel individu. Vous pouvez utiliser la classe Individu ou une autre solution de votre choix.
3. Écrivez une fonction qui génère un individu aléatoirement, c'est-à-dire, un chemin aléatoire valide (qui visite toutes les villes une seule fois).
4. Écrivez une fonction qui affiche un individu. Vous vous en servirez pour contrôler et débuger votre programme par la suite. Affichez toutes les informations qui vous semblent utiles.
5. Écrivez une fonction qui calcule le score d'un individu, c'est-à-dire, le coût du chemin qu'il représente. Vous utiliserez donc le tableau des distances.



6. Écrivez une fonction qui génère une population aléatoire, c'est-à-dire un tableau de N individus générés aléatoirement. Je vous propose d'utiliser la classe *ArrayList* de Java, un tableau dynamique, pour représenter la population (pour faciliter la question suivante). Cette fonction servira d'étape d'initialisation de votre algorithme génétique.
7. Écrivez une fonction qui permet de classer les individus en fonction de leur score. Je vous propose d'utiliser la classe *ArrayList* de Java pour représenter la population. Si de plus vous utilisez la classe *Individu* et implémentez la fonction *compareTo* (Voir doc Java : *Comparable*), vous pourrez utiliser le tri de Java ainsi :

```
ArrayList<Individu> population = new ArrayList();
population.add(....);              /* remplissez avec vos individus */
Collections.sort(population);      /* la population est trie !!! */
```

8. Écrivez une fonction de croisement qui à partir de deux individus, crée un nouvel individu. Sélectionnez aléatoirement le point de pivot, recopiez le chemin du premier individu jusqu'au point de pivot, puis complétez le chemin avec les villes du deuxième individu. Attention, pour que le chemin reste valide, il ne faut pas forcément recopier l'autre moitié du deuxième individu ...
9. Écrivez une fonction qui applique une mutation sur un individu, c'est-à-dire une légère modification aléatoire du chemin (qui doit rester valide).
10. Écrivez maintenant une fonction qui effectue un cycle de l'algorithme génétique. Pour une population donnée, calculez le score de chaque individu, sélectionnez la meilleure partie de la population (par exemple 30%), croisez et mutez aléatoirement votre sélection pour générer une nouvelle population de taille égale à la précédente.
11. Écrivez enfin la fonction qui résout le problème du voyageur de commerce. Dans un premier temps en effectuant un nombre cycles fixe. Comparez votre résultat à la fonction de brute force, testez différents paramètres de votre algorithme (nombre de cycles, taille de la population, taille de la sélection, probabilité de mutation ...).
12. Il existe de nombreuses améliorations à apporter à cet algorithme simple. Cherchez dans des articles scientifique, proposez vos solutions, discutez-en avec vos camarades ...