TP 2 – Algorithmes génétiques

Le but de ce TP va être d'utiliser les algorithmes génétiques pour résoudre le problème du voyageur de commerce.

Le travail est individuel et donnera lieu à une note. Il sera réalisé en python.

1 Le voyageur de commerce

Selon `http://fr.wikipedia.org/wiki/Problème_du_voyageur_de_commerce` :

> L'énoncé du problème du voyageur de commerce est le suivant : étant donné *n* points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point (et revienne au point de départ).
>
> Ce problème est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on devra donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire : Le nombre de chemins possibles passant par 69 villes est déjà un nombre de 100 chiffres. Pour comparaison, un nombre de 80 chiffres permettrait déjà de représenter le nombre d'atomes dans tout l'univers connu !
>
> Ce problème peut servir tel quel à l'optimisation de trajectoires de machines-outils par exemple, pour minimiser le temps total que met une fraiseuse à commande numérique pour percer *n* points dans une plaque de tôle. [...]
>
> Plus généralement, divers problèmes de recherche opérationnelle se ramènent au voyageur de commerce. Un voyageur de commerce peu scrupuleux serait intéressé par le double problème du chemin le plus court (pour son trajet réel) et du chemin le plus long (pour sa note de frais).

2 Démarche proposée

1. Définir, en python, des classes permettant de représenter les concepts suivants :
   - Une *ville* est un point sur un plan (coordonnées x-y)
   - Un *problème* de taille *n* est un ensemble de *n* villes
   - Une *solution* à un problème *P* est un trajet passant une et une seule fois par les *n* villes de *P*. La solution devra être codée de manière à se prêter aux croisements et mutations de votre algorithme génétique.

   - Une *population* est un ensemble de solutions. Dans ce contexte, les solutions sont également appelées *individus*.
2. En vous inspirant de `\\labinfo\dfs\Cours\IARTI\TPs\Voyageur\Exemple.py`, mettez en place une interface graphique (élémentaire !) permettant d'entrer un problème à la souris.
3. Coder des opérateurs de mutation et de croisement adéquats pour vos solutions (attention ! le résultat de ces opérateurs doit toujours être une solution, et donc passer par les *n* villes du problème !)
4. Coder le processus d'évolution de votre population (alternance de sélections, croisements et mutations). Le processus devra être capable de s'arrêter après un nombre donné de générations n'amenant aucune amélioration.
5. Mettre en place le programme principal :
   a) Initialisation du problème (à la souris)
   b) Initialisation de la population
   c) Évolution de la population
   d) Arrêt lorsque qu'aucun progrès n'a été observé durant un certain nombre de générations.

3 Paramètres

Le programme devra contenir un certain nombre de paramètres facilement ajustables (constantes dans le code). Au minimum :

1. La taille de la population
2. La probabilité de mutation d'un individu
3. Le taux de renouvellement de la population (quelle proportion est sélectionnée d'une génération à l'autre, avant de compléter la nouvelle génération par des croisements)
4. Le nombre de générations sans changement avant l'arrêt de la recherche.

Une fois le programme implémenté, vous chercherez une combinaison de ces valeurs donnant de bons résultats pour votre algorithme.

4 Conseils

- L'efficacité de la recherche dépend beaucoup des opérateurs de croisement et de mutation implémentés. On obtient par exemple des résultats raisonnables (mais améliorables !) avec les opérateurs suivants :

  Mutation inversion au hasard d'une portion de chemin

  Croisement En partant d'une ville au hasard, considérer la ville suivante dans chacun des parents et choisir la plus proche. Si celle-ci est déjà présente dans la solution, prendre l'autre. Si elle est aussi déjà présente, choisir une ville non présente au hasard.

- La recherche peut aussi être accélérée en partant de solutions pas trop mauvaises. On peut partir de solutions dites "gourmandes" (*greedy*), qui sont construites en partant d'une ville au hasard et prenant comme suivante la ville la plus proche non encore visitée.

- On obtient également de bons résultat en rajoutant à la sélection une touche “élitiste” : le meilleur individu survivra toujours dans la génération suivante. On évite ainsi de “revenir en arrière”.
- Les personnes désirant une solution *très* efficace pourront s’inspirer des opérateurs présentés dans `http://www.gcd.org/sengoku/docs/arob98.pdf`, qui sont ceux utilisés dans l’applet `http://www-cse.uta.edu/~cook/ai1/lectures/applets/gatsp/TSP.html`.

5 Organisation

- Les cours IARTI des 18.1.07, 25.1.07 et 15.2.07 seront consacrés à ce TP.
- Le travail devra être rendu jusqu’au dimanche 18.1.07 au plus tard.
- Il sera envoyé par email à l’adresse `matthieu.amiguet@he-arc.ch`.
- À rendre :
  - Le programme lui-même, avec les paramètres réglés au mieux que vous pourrez.
  - Une brève description de l’utilisation du programme.
  - Une brève description des opérateurs de mutation et de croisement que vous aurez implémenté ainsi que la description d’éventuels autres points intéressants de votre programme.

6 Évaluation

Votre travail sera évalué selon la grille suivante :

Critère	Coefficient
Traitement du problème (implémentation des classes de base, choix des opérateurs, algorithme génétique proprement dit, …)	3
Efficacité de la résolution du Pb (taille atteignable pour les problèmes, rapidité, …)	2
Maîtrise du langage	2
Respect des consignes (délais, délivrables, …)	1