# Projet “Algorithmes génétiques”

**Attention** Veuillez **lire attentivement l'entier de ce document** qui contient des informations importantes pour le bon déroulement du projet.

## 1 Le projet

Le but de ce projet va être d'utiliser les algorithmes génétiques pour résoudre (de manière approchée) le **problème du voyageur de commerce**.

Le travail sera réalisé par groupes de deux, en python, et donnera lieu à une note.

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 Cahier des charges

Votre programme sera rendu dans un *unique* module python, donc *un seul* fichier nommé selon vos noms de famille (`<VosNoms>.py`)[1]. Ce module contiendra une fonction

[1] Si vous tenez à séparer votre source en plusieurs fichiers, vous pouvez éventuellement rendre un *package* python contenant plusieurs modules.

```
ga_solve(file=None, gui=True, maxtime=0)
```

Cette fonction permettra de résoudre le problème du voyageur de commerce en utilisant un algorithme génétique et fonctionnera de la manière suivante :

- Si l'argument `file` est `None`, on commencera par afficher une fenêtre dans laquelle on peut entrer des points à la souris pour construire un problème. Sinon, cet argument doit être le nom d'un fichier qui contient les coordonnées de villes au format suivant :

  ```
  nom1 x1 y1
  nom2 x2 y2
  ...
  ```

  Les coordonnées *x* et *y* seront des entiers entre 0 et 500.
- Si l'argument gui est à `True`, on devra pouvoir suivre l'évolution de l'algorithme en affichant en cours de déroulement la meilleure solution actuellement trouvée. Sinon, aucun affichage ne sera réalisé.
- Si l'argument `maxtime` est à une valeur positive, la fonction devra retourner une solution après un temps maximum de `maxtime` secondes. Sinon, la fonction devra s'arrêter lorsqu'on constate une stagnation de l'amélioration des solutions.
- La fonction retournera deux valeurs
  - La première valeur est la distance totale à parcourir dans la meilleure solution trouvée ;
  - La deuxième valeur est la liste des noms des villes dans l'ordre où il faut les parcourir.

Votre module devra pouvoir être importé pour pouvoir utiliser la fonction `ga_solve` depuis un autre programme ; on devra également pouvoir le lancer comme programme indépendant. L'appel se fera alors sous la forme

```
python <VosNoms>.py [--nogui] [--maxtime s] [filename]
```

Si `filename` est donné, le problème sera lu depuis ce fichier et sinon il sera construit à la souris ; les options `nogui` et `maxtime` ont une signification évidente.

## 3 Démarche proposée

1. Installez `pygame`[2], qui vous permettra d'utiliser le code d'interface graphique fourni.
2. Si vous travaillez sur une architecture x86, installez `psyco`, qui augmentera considérablement la vitesse d'exécution de votre code (cf. section 5 ci-dessous).
3. Représentez en python les concepts suivants :
   - Une *ville* est un point sur un plan (coordonnées x-y) portant un nom
   - 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*.
4. En vous inspirant de du fichier `GUI_example.py` fourni sur le serveur, mettez en place une interface graphique (élémentaire !) permettant d'entrer un problème à la souris.
5. Implémentez la lecture de problème depuis un fichier. Quelques exemples de fichiers se trouvent dans le répertoire `Data`.

[2] `http://www.pygame.org/`.

6. Codez 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 et revenir au point de départ !)
7. Codez le processus d'évolution de votre population (alternance de sélections, croisements et mutations ; critère d'arrêt).
8. Testez que votre programme fonctionne avec le testeur automatique fourni sur le serveur et qui sera utilisé pour la correction (`PVC-tester.py`).
9. Lorsque tout marche de manière satisfaisante, profilez et optimisez votre code.

# 4 Conseils sur les algorithmes génétiques

– 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.mac.cie.uva.es/~arratia/cursos/UVA/GeneticTSP/JAVA-Simultn/TSP.html`.
– Attention ! plus vos opérateurs de croisement et mutation seront "orientés" plus votre sélection devra être "prudente" pour éviter les maxima locaux !

# 5 Conseils sur l'efficacité

## 5.1 Psyco

Si vous travaillez sur une architecture compatible 386 (quel que soit le système d'exploitation), installez psyco `http://psyco.sourceforge.net`.

Ensuite, ajoutez au début de votre code

```
try:
    import psyco
    psyco.full()
except ImportError:
        pass
```

Cela vous prendra 1 minute ; si psyco n'est pas installé, ça ne changera rien ; et si psyco est installé, vous pouvez obtenir des gains de performance *très* considérable (probablement de l'ordre de 3-4 fois plus vite – voire 10 ou même plus !)

### 5.2 Profilage et optimisation

1. Commencez par implémenter votre algorithme correctement et proprement.
2. Ensuite, profilez-le. Une manière simple de le faire est d'appeler votre programme de la manière suivante

   ```
   python -m cProfile <programme>
   ```
3. Repérez quels sont les endroits qui prennent du temps et optimisez-les.
4. S'il vous reste du temps, reprenez au point 2 !

Quelques points de départ utiles pour optimiser du code python :
- `http://wiki.python.org/moin/PythonSpeed/PerformanceTips` (un peu dépassé, mais contient toujours des information intéressantes)
- `http://jaynes.colorado.edu/PythonIdioms.html#idioms_efficient`

## 6 Contraintes

- Votre programme s'appuyera sur les modules standard python, plus `pygame` et `psyco`. L'éventuelle utilisation d'autres modules tiers doit impérativement être discutée avec le professeur.
- Vous développerez vous-même votre code ! la copie d'une (petite) portion de code pré-existant est tolérable si
  - elle reste occasionnelle et largement minoritaire ;
  - elle est clairement signalée par un commentaire adéquat (avec la source !).

## 7 Organisation

- Le projet sera réalisé par groupes de **deux personnes**. Les groupes devront être **différents** pour chaque projet du module de programmation avancée.
- Le projet est à rendre pour le **vendredi 3 mars 2009**.
- Il sera envoyé par mail à l'adresse `matthieu.amiguet@he-arc.ch`.
- À rendre :
  - Votre programme, dans un unique fichier source python nommé par les noms des deux participants (ex. `DupontDupond.py`) [3]
  - Une brève documentation (docstring ou fichier README) sur les opérateurs et sélection implémentés – et sur d'éventuels autres aspects notables de votre travail.

## 8 Évaluation

Le projet donnera lieu à une note. Il sera évalué selon les critères suivants :
- Efficacité (ie qualité de la solution trouvée dans un temps donné)
- Algorithme génétique (compréhension, qualité de l'implémentation, opérateurs implémentés, ...)
- Qualité générale du projet (maîtrise et clareté du code, commentaires, respect du cahier des charges, ...) ;

Comme d'habitude, merci de faire particulièrement attention au nettoyage de votre code avant de le rendre !

[3] Ou éventuellement dans un *package* python portant ce nom.