Le Problème du voyageur de commerce

Fonctions scilab suceptibles d'être utiles pour ce TP		
diag	grand	sum
sort	find	zeros
gsort	setdiff	

Introduction

Un voyageur de commerce doit se rendre à n villes. Il souhaite le faire en parcourant le moins de distance possible sans repasser deux fois par une même ville sauf pour la première puisqu'il souhaite (à priori !) revenir chez lui.
Ce problème est, contrairement aux apparences, très difficile à résoudre. Il faut en effet des milliards d'années pour trouver la meilleure solution pour 30 villes.
Dans la pratique le problème est encore plus complexe puisqu'il faut tenir compte, en plus de la distance, du coût réel du trajet (route abrupte, ou coût financier : péage ...).
Pour toutes ces raisons, on se contetera de trouver une "bonne" solution au problème. Pour ce faire nous allons utiliser un algorithme génétique.

Algorithmes Génétiques

Ces algorithmes sont inspirés de la théorie de l'évolution et des principes de la génétique.
On part donc d'un ensemble de solutions du problème qu'on essaie d'améliorer en les modifiant par "*croisement*" et par "*mutation*".
Afin de ne pas grossir l'ensemble inutilement, on "*sélectionne*" les meilleures solutions, les plus "*adaptées*".
On voit ainsi que l'on retouve le vocabulaire et les différentes phases de la théorie de l'évolution (et les principes de la génétique) :

Individu	$\longrightarrow$	Une solution du problème.
Population	$\longrightarrow$	Tous les individus, toutes les solutions.
Reproduction	$\longrightarrow$	Croisement de deux individus, deux solutions, pour en produire de nouveaux.
Mutation	$\longrightarrow$	Modification aléatoire d'un individu, d'une solution.
Adaptation	$\longrightarrow$	Evaluation d'un individu, d'une solution (par rapport au problème à résoudre)
Selection	$\longrightarrow$	Elimination des individus, des solutions les moins adaptés.

Application

On mettra en pratique dans ce TP un algorithme génétique pour la résolution du problème du voyageur de commerce. On trouvera, bien entendu, **une solution** qui ne sera pas nécéssairement la meilleure.
Les n villes à parcourir seront numérotées de 1 à n et on considère que le voyageur partira toujours de la ville 1 (chez lui) et qu'il y retournera à la fin de son périple.

Ecrire une fonction scilab, **Carto** prenant comme paramètre d'entrée le nombre n de villes à parcourir et donnant en sortie une matrice aléatoire nxn, D, représentant les distances entre les villes, Il s'agira de notre carte. L'élément d_{ij} de D représentera ainsi la distance entre les villes i et j.

Indications et remarques

La matrice doit être symétrique à diagonale nulle (la distance des villes i à j est évidemment la même qu'entre j et i ; la distance de i à i est nulle).
Si A est une matrice quelconque alors la matrice $A + {}^tA$ est symétrique.

Ecrire une fonction scilab, **Populat**, prenant comme paramètres d'entrée le nombre n de villes et le nombre $2m$ de solutions (individus) que l'on souhaite générer et qui renvoie une matrice, P, de solutions aléatoires. Il s'agira de notre population de départ.

Indications et remarques

La matrice sera donc de taille $(n, 2m)$. Chaque colonne est une solution possible dont le premier élément sera toujours 1 (ville de départ imposée) et les autres des permutations de $\{2, \ldots, n\}$.

Ecrire une fonction scilab, **CalculAdapt**, qui prend comme paramètre d'entrée une solution (un individu) et qui renvoie la distance totale parcourue si l'on suit ce trajet.

Indications et remarques

Il ne faut pas oublier le dernier trajet, de la dernière ville à la première (pour rentrer à la maison !).
Pour calculer la distance si l'on suit le trajet $(1, 4, 3, 2)$ il suffira de faire $d_{14} + d_{43} + d_{32} + d_{21}$.
Les matrice D des distances sera supposée déjà crée, cette fonction prendra donc en paramètre d'entrée une solution et utilisera D pour calculer son "adaptation".
On pourra modifier cette fonction pour calculer les distances des $2m$ solutions, la fonction prendra alors la matrice des $2m$ solutions en entrée et donnera en sortie un vecteur de taille $2m$ contenant les distances des $2m$ solutions.

Ecrire une fonction **SelectElit** qui prend en paramètre une population P de $2m$ individus (au début de l'algorithme la population sera donnée par la fonction **Populat**) et qui renvoie les m meilleures solutions.

Indications et remarques

La fonction **SelectElit** utilisera bien-sûr la fonction **CalculAdapt**. Cette sélection est dite élitiste puiqu'elle ne conserve que la meilleure moitié de l'ensemble des solutions.

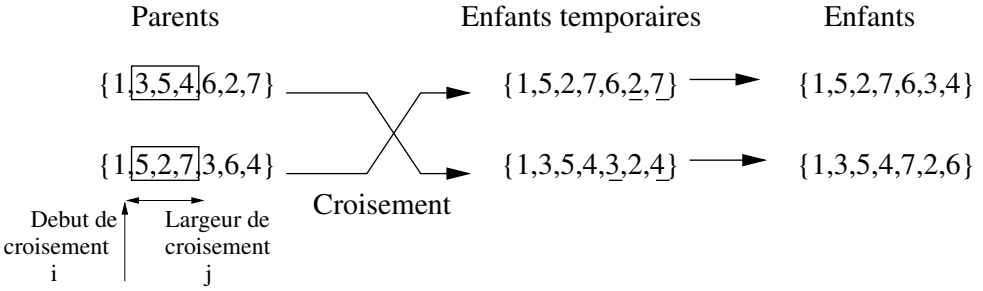
Ecrire une fonction **SelectTourn** qui prend en paramètre une population P de $2m$ individus et qui renvoie m solutions. Dans cette fonction on choisira au hasard 2 solutions parmi les $2m$ et ne conservera que la meilleure des deux, on répetera cette élimination m fois. Cette fonction donnera donc en sortie m solutions parmi les $2m$ initiales.

Indications et remarques

La fonction, **SelectTourn**, met en oeuvre une méthode par tournoi : on choisit deux individus, on les fait combattre et on ne conserve que le meilleur.
Il est possible qu'un individu (solution) participe à plusieurs tournois et ainsi être conservé en plusieurs exemplaires (s'il gagne). Il peut donc y avoir parmi les m solutions en sortie de cette fonction plusieurs solutions identiques.

Croisement

Après avoir sélectionné les m individus les plus adaptés, on répartit ces survivants en couple aléatoire que l'on croise pour obtenir deux individus (enfants).
Le croisement consiste à échanger des parties des parents, on obtient ainsi des enfants temporaires dans lesquels il peut y avoir des villes en double. Il faut alors remplacer les doublons par les villes manquantes en respectant l'ordre du parent n'ayant pas donné la partie croisée.



Dans l'exemple ci-dessus on a croisé à partir de $i = 2$ et pour une largeur de $j = 3$. Les enfants temporaires ainsi obtenus possèdent des doublons, dans le premier enfant par exemple on retrouve deux fois les villes 2 et 7, il convient alors de les remplacer par les villes manquantes (ici 3 et 4). On ajoute les villes manquantes dans le même ordre que celui où elles apparaissent dans le parent "principal" (celui donnant la partie non croisée : sur le graphique il est sur la même ligne). Attention, on ne modifie pas la partie croisée.

Ecrire une fonction **Croisement** qui prend en paramètre d'entrée 2 individus et qui renvoie deux enfants obtenus par le procédé décrit précédemment.

Indications et remarques

Les entiers i, de début de croisement, et j de largeur de croisement sont choisis aléatoirement, $i \in 2, \ldots n-1$ (la première ville du parcours sera toujours à 1) et $j \in 1, \ldots, n-i$.
Les parties obtenues par croisement ne doivent pas être altérées.
On peut remarquer que les parties croisées permettent de déterminer les éléments en double et ceux manquant. Il suffit alors de chercher l'indice des éléments en double en dehors de la plage d'indice du croisement et de les remplacer par les éléments manquants (dans l'ordre du parent !).
Cette fonction est assez délicate, on fera des tests simples pour s'assurer de son bon fonctionnement.

Ecrire une fonction **Mutation** qui prend en entrée un individu et qui renvoie un autre obtenu par mutation. On se limitera au cas le plus simple : une mutation permutera au hasard deux villes de l'individu en entrée (à l'exclusion de la ville 1).

Indications et remarques

Cette fonction est particulièrement simple puisqu'elle se résume au tirage aléatoire de deux entiers, i et j, (indices) entre 2 et n et à l'échange des villes en positions i et j.

Ecrire la fonction principale **Genetiq** qui prend en entrée un entier n (nombre de villes), le nombre d'individus, $2m$, dans la population initiale, le taux, τ, de la population subissant une mutation à chaque itération, la méthode de sélection choisie (élitiste ou par tournoi) et enfin le nombre *iter* d'itérations.
Cette fonction donnera en sortie la solution retenue.

Indications et remarques

Le nombre d'itérations est choisi au début par l'utilisateur. Au début de la fonction on crée une population de départ (Génèse) et une fois pour toute une carte (fonction Carto) puis à chaque itération on commence avec les $2m$ solutions (du début pour la première itération puis avec les solutions de l'itération précédente) que l'on sélectionne par la méthode retenue (élitiste ou par tournoi). On croise ensuite les m solutions retenues pour revenir à $2m$ solutions. On fait enfin muter τ de ces solutions. A la dernière itération on sélectionne la meilleure solution.