TP Optimisation - Globale

Exercice 1 : Fonction Triviale

On considère une fonction $f(\vec{x})$ que l'on cherche à optimiser sur $\{0,1\}^n$. La fonction f est définie de la façon suivante :

$$f(\vec{x}) = \sum_{i=1}^{i=n} x_i$$

Donner la modélisation mathématique de ce problème (objectif, état contraintes).

On se propose de maximiser cette fonction à l'aide d'un schéma de recuit. On prendra $n = 1000$.

On utilisera l'algorithme suivant :

CHAUFFAGE
$T = 0.1$;
nbTransitions=1000 ;
DO {
    acceptCount=0 ;
    T=T*1.1 ;
    FOR (i=0 ;i<nbTransitions ;i++)
        initAleatEtat($\vec{x}_i$)
        calculCritere $y_i = f(\vec{x}_i)$
        $\vec{x}_j$=genererVoisin($\vec{x}_i$) ;
        calculCritere $y_j = f(\vec{x}_j)$
        IF accept(yi,yj,T,minimisation)
            acceptCount++ ;
        END IF
    END FOR
    χ_0 =acceptCount/nbTransitions ;
} **WHILE** ($\chi_0 < 0.8$)
$T_{init} = T$

REFROIDISSEMENT
$\alpha = 0.995$;
initAleatEtat($\vec{x}_i$)
calculCritere $y_i = f(\vec{x}_i)$
$T = C_{init}$
DO {
    FOR (i=0 ;i<nbTransitions ;i++)
        $\vec{x}_j$=genererVoisin($\vec{x}_i$) ;
        calculCritere $y_j = f(\vec{x}_j)$
        IF accept(yi,yj,T,minimisation)
            $x_i = \vec{x}_j$
            $y_i = y_j$
        END IF
    END FOR
    $T = T * \alpha$;
    Afficher T, $\vec{x}_i$, y_i
} **WHILE** ($T > \varepsilon * T_{init}$)

La méthode **accept** possède un paramètre binaire (**minimisation**) permettant de sélectionner un problème de minimisation ou de maximisation.

Dans le code java, on modifie aléatoirement le point $\vec{x}_i$ (**xi.genererVoisin**) et on évalue le critère associé (y_j). S'il est accepté on ne fait rien, sinon on replace $\vec{x}_i$ dans sa position initiale (opérateur **xi.comeBack**). Cette méthode évite des copies coûteuses surtout pour des espaces de grande dimension mais nécessite de mémoriser les modifications de l'espace d'état afin de pouvoir revenir au point initial après avoir généré un voisin.

Rappel : pour compiler un programme java (Recuit.java) dans un terminal, lancer la commande **javac Recuit.java** ; pour l'exécution **java Recuit**.

Pour éditer un code source vous pouvez utiliser un éditeur ; par exemple **emacs**.

Écrire le modèle mathématique associé à ce problème (espace d'état, fonction objectif, contrainte).

Ecrire le code de la fonction **accept**.

Compléter la classe **Etat.java** afin de résoudre ce premier problème.

Exercice 2 : Sac à dos

On considère un ensemble de n objets dont les poids et les valeurs sont tirés aléatoirement entre 1 et 100. On désire optimiser la valeur d'un sac pour lequel la capacité maximum est de 2000.

Donner la modélisation mathématique de ce problème (objectif, état contraintes).

Adapter le programme précédent afin de trouver la solution la plus efficace pour remplir le sac.

On prendra en compte la contrainte dans la fonction objectif.

Modifier la méthode **calculCritere()** dans la classe **Etat.java**. A chaque palier de température on affichera T, $\vec{x}_i$, y_i, le poids du sac ainsi que sa valeur.

La classe **Data.java** permet de générer aléatoirement ces "n" objets (méthode **genererObjets(int n)** qui sera appelée en préfixant le nom de la classe : **Data.genererObjets(n) ;** car la méthode est statique). Les poids et les valeurs des objets sont contenues dans les variables **Data.tabPoids[]** et **Data.tabValeurs[]**.

Exercice 3 : TSP

Cet exercice consiste à résoudre le problème du voyageur de commerce (TSP) sur un ensemble de n points. On dispose aléatoirement un ensemble de n points sur un cercle de rayon $R = 100$ (méthode **Data.genererVillesCercle(n)**). On place ensuite les coordonnées cartésiennes de chaque point dans un tableau à deux dimensions (variable **Data.tabVilles[][]**).

$coord_x$	x_1	x_2	x_3	...	x_n
$coord_y$	y_1	y_2	y_3	...	y_n

On considère le vecteur d'état initial suivant :

$$\vec{X_0} = \boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{...}\boxed{n}$$

auquel on associe la distance :

$$d(\vec{X_0}) = \sum_{i=1}^{i=n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} + \sqrt{(x_n - x_1)^2 + (y_n - y_1)^2}$$

Donner la modélisation mathématique de ce problème (objectif, état contraintes).

On propose d'utiliser l'opérateur de voisinage consistant à permuter deux villes sélectionnées aléatoirement dans le tableau . Ecrire les méthodes de la classe **Etat.java** pour ce problème.

Modifier la méthode **calculCritere()** dans la classe **Etat.java**.

Adapter les paramètres de votre code afin de résoudre efficacement ce problème du voyageur de commerce pour lequel on connaît la longueur du tour optimal ($2.\pi.R$). On prendra $n = 100$ et $R = 100$.

Comparer cet opérateur de voisinage avec celui consistant à permuter l'ensemble des villes entre deux positions aléatoires.

Exercice 4 : Séquencement des avions à l'atterrissage

On considère $n = 100$ avions devant atterrir sur une piste d'aéroport et on se propose d'optimiser l'ordre de séquencement à l'arrivée. Suivant la catégorie de sillage des aéronefs, il est nécessaire d'assurer une distance minimum de séparation à l'approche. On utilisera les séparations suivantes :

$$[s_{ij}] = \begin{bmatrix} 4 & 5 & 6 \\ 3 & 3 & 4 \\ 3 & 3 & 3 \end{bmatrix}$$

(i leader, données en NM), $i, j = 0, 1, 2$; "0" Heavy ; "1" Medium ; "2" Light)

L'objectif est de minimiser la somme des distances de séparation sur l'ensemble du segment de la finale. Chaque avion ayant des contraintes de vitesse, on fait l'hypothèse que sa position dans la séquence ne peut pas changer de plus de 4 rangs (en avant ou en arrière).

Donner la modélisation mathématique de ce problème (objectif, état contraintes).

La méthode **Data.genererAvions(int n)** permet de générer une séquence aléatoire dans le tableau **Data.tabAvions[]** ; les séparations sont contenues dans le tableau **Data.separation[][]**.

Modifier le programme pour traiter ce nouveau problème.