Corrigé TD 4

Sol exo 1

Pour le probleme TSP symetrique métrique (int k , int [][] d), une solution est une permutation des n entiers x=(xi) xi∈N , 0≤xi≤k-1 ; xi≠xj pour i≠j,
L'espace de recherche est S = {x} ; |S|= (k-1) ! / 2
Pour k= 5 ; les solutions candidates sont 1 2 3 4 5 ; 1 2 3 5 4 ; 1 2 4 3 5 ; 1 2 4 5 3 ; …………….
Dont le nombre est (5-1)!/2=12.
formulation mathématique :

$$\begin{cases} \min \sum_{i=1}^{k-1} d(x_i, x_{i+1}) + d(x_{k-1}, x_0) \\ \quad x_i \in \mathrm{N};\ \ x_i \neq x_j \ pour\ i \neq j \\ \qquad 0 \leq x_i \leq k-1 \end{cases}$$

Remarque : il y a d'autres formulations en nombres binaires

```
BCOTSP  ( int k , int [][] d) // k nombre de villes , d matrice k×k des distances entre les villes : données
{
//    Parameters
Int n = 5k ;  // taille de la population
Int m = n*0.6 ;   // nombre total de butineuses ; sources sélectionnées
Int e= m*0.3 ;   // nombre de butineuses pour les sources riches en nectar
Int n1 = 10 ;  // scouts (taille du voisinage) pour les sources riches
Int n2 = 3 ;  // scouts (taille du voisinage) pour les sources moins riches
Maxiter = 1000 k ;   // nombre d'itérations
 //-----------------------------------------------------------------------------------------------------

// neighbor : retourne un voisin X (une solution , un tableau) aléatoire de la solution d'indice i dans la population en permutant deux villes aléatoires
int [] neighbor ( int i ) {
  X[]=pop[i] ;
  Ind1 = int(k*rand()) ;
  Ind2 = int(k*rand()) ;
  Tmp= X[ind1] ;
  X[ind1]= X[ind2] ;
  X[ind2]=Tmp ;
  return X ; }
//-------------------------------------------------------------------------------------

// Initialisation, création de la population initiale pop, une matrice n×k
Void initialisation()
{ for(i=0 ; i<n ; i++)
    for(j =0 ; j<k ; j++)
      {do
           ind = Int(k * Rand())
      while  (pop[i][ind] != 0 )    // le tour (le cycle) doit être hamiltonien
      pop [I][ind]= j :}}
//-------------------------------------------------------------------------------------

// Calcul de la longueur de la solution d'indice i dans la population pop
int longueur(int i) {
int s=0:
```

```
for(j=0 ; j<n-1 ; j++)
        s+=d[pop[i][j]] [pop[i][j+1]]
return s+ d[pop[i][n-1]] [pop[i][0]]  }
//------------------------------------------------

// Évaluation des solutions de la population, calcul de leurs fitness, leurs longueurs
Void evaluation()
{  int Sumfitness =0 ;  // à utiliser dans la roulette de sélection
for(i=0 ; i<n ; i++)
   { Fitness[i] = longueur(i) ;  // à utiliser dans la roulette de sélection
     Sumfitness += Fitness[i] ;
     If (fitness[i] < lopt)
          {xopt= pop[i] ; lopt= fitness[i] ;} // Mise à jour de la meilleure solution trouvée jusqu'ici
//------------------------------------------------
// Fonction roulette pour simuler la roue de la fortune dans la sélection des solutions
int roulette ()
{  float s = 0 ;
    float s1 = sumfitness*rand() ; // arrester la roulette
    Int j =0 ;
   while (s < s1)
         {  s+=fitness[j] ; j++;}  // tourner la roulette
    return j ; }
//------------------------------------------------
// Programme principal
Lopt= infini ; xopt= null ;  // solution (cycle) optimale recherchée et sa longueur
initialisation()   // création de la population initiale
evaluation ()   // evaluation des solutions de la population courante

for (iter = 0; iter < maxiter; iter ++)    // boucle principale
  // On construit une nouvelle population pop1 à partir de pop en 3 étapes
{
 // Première étape : les sources riches
 for(i=0 ; i<e ; i++)
       {  r = roulette() ;
          x= pop[r] ;
         for (j=0 ; i<n1 ; i++)   // calcul de n1 voisins de la solution pop[i]
             {  b= neighbor(r) ;  // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                if (longueur(b) < longueur(x))  // recherche du meilleur voisin
                           x = b ;  }
       pop1[i]= x ;   }  // prise du meilleur voisin
}     // fin de la première étape
//----------------------------------------------------------------
// Deuxième étape : les sources moins riches en nectar
 for(i=e ; i<m ; i++)
       {  r = roulette() ;
          x= pop[r] ;
         for (j=0 ; i<n2 ; i++)   // calcul de n2 voisins de la solution pop[i]
              { b= neighbor(r); // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                 if (longueur(b) < longueur(x))  // recherche du meilleur voisin
                           x = b ;  }
       pop1[i]= x ;   }  // prise du meilleur voisin
      }  // fin de la deuxième étape
//------------------------------------------------------------------------------------------
// Troisième étape : on remplace les n-m solutions restantes par de nouvelles solutions aléatoires
for(i=m ; i<=n-1 ; i++)
```

```
   for(j =0 ; j<k ; j++)
       {do
          ind = Int(k * Rand())
       while  (pop1(I,ind) !=0)
       pop1 [i][ind]= j}   // fin de la troisième étape
//-------------------------------------------------------------------------------------
pop = pop1 ;  //  la population courante dévient pop1
evaluation ()   // evaluation des solutions de la population courante
} // fin de la boucle principale
printf( xopt , fopt) ]   }
```

===

Solution exo 2

Minimum de la fonction de test de De Jong f (x) = $\sum_{i=1}^{k} {x_i}^2 \quad -5.12 <= xi <= 5.12$ en appliquant BCO

Une solution candidate de ce problème est un vecteur x de n réels xi ; i=1,n ; xi∈[-5.12,+5.12] ; l'espace de recherche est S = [-5.12,+5.12] ⊂R ; |S|= ∝ (il s'agit d'un problème d'optimisation continue).
Le minimum exact de la fonction de DE JONG est 0 en (0,0,…,0).

```
JONGBCO (int k )
{
// PARAMETRES
Int n = 5k ;  // taille de la population
Int m = n*0.6 ;   // nombre total de butineuses ; sources sélectionnées
Int e= m*0.3 ;   // nombre de butineuses pour les sources riches en nectar
Int n1 = 10 ;  // scouts (taille du voisinage) pour les sources riches
Int n2 = 3 ;  // scouts (taille du voisinage) pour les sources moins riches
Maxiter = 1000 k ;   // nombre d'itérations
 //--------------------------------------------------------------------------------------------------

// neighbor : retourne un voisin X (une solution , un tableau de réels) aléatoire de la solution d'indice i dans la
population pop en permutant deux composantes (coordonnées) aléatoires
float [] neighbor ( int i ) {
  X[]=pop[i] ;
  Ind1 = int(k*rand()) ;
  Ind2 = int(k*rand()) ;
  Tmp= X[ind1] ;
  X[ind1]= X[ind2] ;
  X[ind2]=Tmp ;
  return X ; }
//-------------------------------------------------------------------------------------

// Initialisation, création de la population initiale pop, une matrice n×k
Void initialisation()
{ float pop [][] ;
  for(i=0 ; i<n ; i++)
     for(j =0 ; j<k ; j++)
        pop [i][j]= -5.12+10.24*rand() ;}
//-------------------------------------------------------------------------------------

// Calcul de la fitness (la valeur de la fonction f) de la solution d'indice i dans la population pop
int f(int i) {
```

```
float s=0:
for(j=0 ; j<k ; j++)
        s+ =  pop[i][j] * pop[i][j] ;
return s ;  }
//--------------------------------------------------

// Évaluation des solutions de la population, calcul de leurs fitness ;
Void evaluation()
{  float sumfitness =0 ;  // à utiliser dans la roulette de sélection
for(i=0 ; i<n ; i++)
   { fitness[i] = f(i) ;  // à utiliser dans la roulette de sélection
     sumfitness += fitness[i] ;
     If (fitness[i] < fopt)
          {xopt= pop[i] ; fopt= fitness[i] ;} // Mise à jour de la meilleure solution trouvée jusqu'ici
//--------------------------------------------------
// Fonction roulette pour simuler la roue de la fortune dans la sélection des solutions
int roulette ()
{  float s = 0 ;
    float s1 = sumfitness*rand() ; // faire arreter la roulette
    Int j =0 ;
   while (s < s1)
         {  s+=fitness[j] ; j++;}  // faire tourner la roulette
    return j ; }
//--------------------------------------------------

// Programme principal
xopt= null ; fopt= infini ;  // solution (vecteur) optimal recherché et le min de f
initialisation()   // création de la population initiale
evaluation ()   // evaluation des solutions de la population courante

for (iter = 0; iter < maxiter; iter ++)    // boucle principale
  // On construit une nouvelle population pop1 à partir de pop en 3 étapes
{
 // Première étape : les sources riches
 for(i=0 ; i<e ; i++)
      {  r = roulette() ;
          x= pop[r] ;
         for (j=0 ; i<n1 ; i++)   // calcul de n1 voisins de la solution pop[i]
             {  b = neighbor(r) ;  // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                 if (f(b) < f(x))  // recherche du meilleur voisin
                             x = b ;  }
       pop1[i]= x ;   }  // prise du meilleur voisin
}     // fin de la première étape
//-------------------------------------------------------------
// Deuxième étape : les sources moins riches en nectar
 for(i=e ; i<m ; i++)
      {  r = roulette() ;
          x= pop[r] ;
         for (j=0 ; i<n2 ; i++)   // calcul de n2 voisins de la solution pop[i]
              { b= neighbor(r); // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                  if (f(b) < f(x))  // recherche du meilleur voisin
                             x = b ;  }
       pop1[i]= x ;   }  // prise du meilleur voisin
      }  // fin de la deuxième étape
//------------------------------------------------------------------------------------------
```

```
// Troisième étape : on remplace les n-m solutions restantes par de nouvelles solutions aléatoires
for(i=m ; i<=n-1 ; i++)
        for(j =0 ; j<k ; j++)
                pop1 [i][j]= -5.12+10.24*rand() ;}  // fin de la troisième étape
//-----------------------------------------------------------------------------------
pop = pop1 ;  //  la population courante dévient pop1
evaluation ()   // evaluation des solutions de la population courante
} // fin de la boucle principale
printf( xopt , fopt) ]   }
```

==

Solution exo 3

Formulation mathématique :

Une solution candidate de ce problème est un vecteur x =(x_1,x_2,…,x_k) de k entiers tel que x_i est le nombre de pièces fabriquées par l'unité i ; l'espace de recherche est S= $\{0,n/k\}^k$; ainsi |S| = $(n/k)^k \approx O(2^k)$.

Le problème s'écrit alors :

$$\begin{cases} \min Cmax = \max\limits_{i=1,k}\{x_i * t_i\} \\ x_i \in N;\ 0 \leq x_i \leq \frac{n}{k} \\ \sum_{i=1}^{k} x_i = n \end{cases}$$

Remarques :

- Si n ≤ k (chaque unité produit une seule pièce au plus xi=1 ou 0), le problème devient « facile » (en O(n)) , la solution est un vecteur binaire de k bits avec Cmax= $\max\limits_{i=1,n}\{t_i\}$.
- Si k=1 (une seule unité, chaque pièce nécessite ti = t unités de temps), le problème devient « facile » (en O(1)) , la solution est n avec Cmax= n ∗ t.
- Si n>k, le problème est NP-difficile.

Implémentation

```
BCOSCHEDULING  ( int n , int k , int t[] ) // k nombre de villes , d matrice k×k des distances entre les villes : données
{
//    Parameters
Int psize = 5k ;  // taille de la population
Int m = n*0.6 ;   // nombre total de butineuses ; sources sélectionnées
Int e= m*0.3 ;   // nombre de butineuses pour les sources riches en nectar
Int n1 = 10 ;  // scouts (taille du voisinage) pour les sources riches
Int n2 = 3 ;  // scouts (taille du voisinage) pour les sources moins riches
Maxiter = 1000 k ;   // nombre d'itérations
 //----------------------------------------------------------------------------------------------

// Initialisation, création de la population initiale pop, une matrice n×k
Void initialisation()
{ for(i=0 ; i<psize ; i++)
      s =0 ;
     for(j =0 ; j<k ; j++)
            r = int(n/k * Rand()) ;   // générer un entier aléatoire compris entre 0 et n/k
            s+=r ;
            if  (s ≤ n)      // satisfaire la 2-ème contrainte
                 pop [i][j]= r ;
```

```
            else
                break ;
        }}
//-----------------------------------------------------------------------------------


// neighbor : retourne un voisin X (une solution , un tableau) aléatoire de la solution d'indice i dans la population en rajoutant  (ou en soustrayant) 1 à une composante aléatoire de ce vecteur
int [] neighbor ( int i )
{     X[]=pop[i] ;
      S=0 ;
      For (j=0; j<k; k++)
            S+=x[j] ;
      If (s+1 ≤ n )
           { Ind = int(k*rand()) ;
            X[ind]= X[ind]+1 ; }
       Else
            X[ind]= X[ind]-1 ;

       return X ; }
//-----------------------------------------------------------------------------------


// Calcul de Cmax de la solution d'indice i dans la population pop
int Cmax (int i) {
int s=0:
for(j=0 ; j<k ; j++)
        if (pop[i][j]*t[j] > s)
                s= pop[i][j]*t[j] ;
return s ;  }
//-----------------------------------------------

// Évaluation des solutions de la population, calcul de leurs fitness, leurs Cmax
Void evaluation()
{  int Sumfitness =0 ;  // à utiliser dans la roulette de sélection
for(i=0 ; i<n ; i++)
   { Fitness[i] = Cmax(i) ;  // à utiliser dans la roulette de sélection
     Sumfitness += Fitness[i] ;
     If (fitness[i] < lopt)
          {xopt= pop[i] ; Cmaxopt= fitness[i] ;} // Mise à jour de la meilleure solution trouvée jusqu'ici
//-----------------------------------------------
// Fonction roulette pour simuler la roue de la fortune dans la sélection des solutions
int roulette ()
{  float s = 0 ;
   float s1 = sumfitness*rand() ; // arrester la roulette
   Int j =0 ;
  while (s < s1)
        {  s+=fitness[j] ; j++;}  // tourner la roulette
   return j ; }
//-----------------------------------------------
// Programme principal
Cmaxopt= infini ; xopt= null ;  // solution (cycle) optimale recherchée et sa longueur
initialisation()   // création de la population initiale
evaluation ()   // evaluation des solutions de la population courante
```

```
for (iter = 0; iter < maxiter; iter ++)    // boucle principale
  // On construit une nouvelle population pop1 à partir de pop en 3 étapes
{
 // Première étape : les sources riches
 for(i=0 ; i<e ; i++)
       {  r = roulette() ;
           x= pop[r] ;
          for (j=0 ; i<n1 ; i++)   // calcul de n1 voisins de la solution pop[i]
              {  b= neighbor(r) ;  // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                 if (Cmax (b) < Cmax(x))  // recherche du meilleur voisin
                              x = b ;  }
        pop1[i]= x ;   }  // prise du meilleur voisin
}     // fin de la première étape
//---------------------------------------------------------------
// Deuxième étape : les sources moins riches en nectar
 for(i=e ; i<m ; i++)
       {  r = roulette() ;
           x= pop[r] ;
          for (j=0 ; i<n2 ; i++)   // calcul de n2 voisins de la solution pop[i]
               { b= neighbor(r); // calcul d'un voisin aléatoire de la solution courante sélectionnée d'indice r
                  if (Cmax (b) < Cmax (x))  // recherche du meilleur voisin
                              x = b ;  }
        pop1[i]= x ;   }  // prise du meilleur voisin
       }  // fin de la deuxième étape
//------------------------------------------------------------------------------------------------
// Troisième étape : on remplace les psize-m solutions restantes par de nouvelles solutions aléatoires
for(i=m ; i<=psize-1 ; i++)
    for(j =0 ; j<k ; j++)
        {do
           ind = Int(k * Rand())
        while  (pop1(I,ind) !=0)
        pop1 [i][ind]= j}   // fin de la troisième étape
//-------------------------------------------------------------------------------------
pop = pop1 ;  //  la population courante dévient pop1
evaluation ()   // evaluation des solutions de la population courante
} // fin de la boucle principale
printf( xopt , fopt) ]   }
```