TP 3 : Algorithme de Little

Esteban Becker

07/04/2023

Table des matières

1 Travail préparatoire

1.1 Compilation et exécution

Le code suivant a été codé en C et compilé pour machine Linux en utilisant gcc via le terminal. Le programme affiche bien les coordonnées des 6 villes.

1.2 Calcul de la distance

Etant dans un espace a 2 dimensions, la distance entre deux points est donnée par la formule suivante : $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ La distance entre les villes 1 et 2 est donc de $d = \sqrt{(2-1)^2 + (2-1)^2} = \sqrt{2}$ De plus, la distance entre la ville i et j est la même que la distance entre la ville j et i. Ainsi la matrice est symetrique, il suffit de calculer la partie triangulaire supérieur ou la partie triangulaire inférieur de la matrice.

Ainsi la fonction de calcul de la distance est la suivante :

```
1  for (int i=0; i<NBR_TOWNS; i++)
2      {
3          int j = 0 ;
4          while (j < i)
5          {
6              // Compute the distance between two towns
7              dist[i][j] = sqrt (pow(coord[i][0]-coord[j][0],2) + pow(coord[i][1]-coord[j
       ][1],2)) ;
8              // Symmetric matrix
9              dist[j][i] = dist[i][j] ;
10             j++ ;
11         }
12         dist[i][i] = -1 ;
13     }
```

Ainsi, à la ligne 7 on calcul la distance entre la ville i et la ville j, et à la ligne 9 on recopie cette distance dans la case correspondante de la matrice symétrique. La ligne 12 permet de mettre une valeur négative dans la diagonale de la matrice, afin de ne pas prendre en compte la distance entre une ville et elle même.

1.3 Création fonction build_nearest_neighbour

Cette fonction permet d'avoir une première approximation de la solution optimale.

```
1  double build_nearest_neighbour()
2  {
3      /* solution of the nearest neighbour */
4      int i, sol[NBR_TOWNS] ;
5
6      /* evaluation of the solution */
7      double eval = 0 ;
8
9      /*initialise solution*/
10     for (i=0; i<NBR_TOWNS;i++){
11         sol[i] = 0;
12     }
13
14     /*find the solution with nearest neighbour*/
15     for (i = 1; i<NBR_TOWNS; i++)
16     {
17         int j, k ;
18         double min = -1 ;
19         for (j=0; j<NBR_TOWNS; j++)
20         {
21             for (k=0; k<i; k++)
22             {
23                 if (sol[k] == j)/*If town already in solution*/
24                     break ;
```

```
25              }
26              if (k<i)
27                  /*If town already in solution*/
28                  continue ;
29              if (min<0 || dist[sol[i-1]][j] < min)
30              {
31                  min = dist[sol[i-1]][j] ;
32                  sol[i] = j ;
33              }
34          }
35      }
36
37      eval = evaluation_solution(sol) ;
38      printf("Nearest neighbour ") ;
39      print_solution (sol, eval) ;
40
41      for (i=0;i<NBR_TOWNS;i++) best_solution[i] = sol[i] ;
42      best_eval  = eval ;
43
44      return eval ;
45  }
```

A la ligne 10 on initialise la solution avec toutes les villes valant 0.

Ensuite dans la boucle for de la ligne 16 à la ligne 30 on cherche la ville la plus proche de la ville précédente, et on l'ajoute à la solution. On commence à partir de i=1 car on commence forcément par la ville 0.

De la ligne 21 à la ligne 28 on vérifie que la ville n'a pas encore été visité. Si elle a déjà été visité, on l'ignore. Sinon à la ligne 29 on vérifie si la ville en cours d'étude est plus intéréseente que les villes précédemment étudier, si oui, on la mémorise puis on étuduie la ville suivante.

2 Algorithme de Little

2.1 Mettre des 0 dans la matrice des distances

Premièrement nous devons faire apparaitre au moins un 0 dans chaque ligne et chaque collone de la matrice des distances.

```
1       /* Make sure that there is a 0 in every row*/
2       for (i = 0; i < NBR_TOWNS; i++)
3       {
4           double min = -1;
5           for (j = 0; j < NBR_TOWNS; j++)
6           {
7               if(d[i][j] < min || min == -1)
8               {
9                   if(d[i][j] != -1)
10                      min = d[i][j];
11              }
12          }
13          for (j = 0; j < NBR_TOWNS; j++)
14          {
15              if(d[i][j] != -1)
16                  d[i][j] -= min;
17          }
18          eval_node_child += min;
19      }
20
21      /* Make sure that there is a 0 in every column*/
22      for (j = 0; j < NBR_TOWNS; j++)
23      {
24          double min = -1;
25          for (i = 0; i < NBR_TOWNS; i++)
26          {
```

```
27              if(d[i][j] < min || min == -1)
28              {
29                  if(d[i][j] != -1)
30                      min = d[i][j];
31              }
32          }
33          for (i = 0; i < NBR_TOWNS; i++)
34          {
35              if(d[i][j] != -1)
36                  d[i][j] -= min;
37          }
38          eval_node_child += min;
39      }
```

Il ne faut pas oublier de mettre à jour le poids de l'évaluation de la solution au ligne 18 et 38. De plus, il ne faut pas oublier que les valeurs "-1" ne doivent pas être prise en compte dans le calcul du minimum.

2.2 Trouvez le 0 avec la plus grosse penalité

Pour effectuer celà nous allons parcourir toute la matrice, et à chaque fois qu'on tombe sur un 0, on va calculer la penalité de ce 0. La penalité d'un 0 est la somme des plus petites valeurs de la ligne et de la colonne du 0. Il faut penser à exlure le 0 de la recherche

```
1       for(i = 0; i < NBR_TOWNS; i++)
2       {
3           for(j = 0; j < NBR_TOWNS; j++)
4           {
5               if (d[i][j] == 0)
6               {
7                   double penalty_x = 0 , penalty_y = 0 ;
8                   double taken_x = 0 , taken_y = 0 ;
9                   int k ;
10                  /* Compute the penalty of the zero */
11                  for (k = 0; k < NBR_TOWNS; k++)
12                  {
13                      if (k != i && d[k][j] < penalty_x && d[k][j] >= 0){
14                          penalty_x = d[k][j] ;
15                          taken_x = 1;
16                      }
17                      if (k != j && d[i][k] < penalty_y && d[i][k] >= 0){
18                          penalty_y = d[i][k] ;
19                          taken_y = 1;
20                      }
21                  }
22                  double penalty = penalty_x + penalty_y ;
23
24                  /* Update the zero with the max penalty */
25                  if (penalty > max_penalty && taken_x == 1 && taken_y == 1)
26                  {
27                      max_penalty = penalty ;
28                      izero = i ;
29                      jzero = j ;
30                  }else if(taken_x == 0 || taken_y == 0){
31                      max_penalty = INFINITY ;
32                      izero = i ;
33                      jzero = j ;
34                  }
35              }
36          }
37      }
```

Il faut penser lors de la vérification de la condition aux lignes 13 et 17 à exclure le 0 de la recherche et a ne pas prendre en compte les chemins impossibles. De plus, il faut penser au cas où il y a un zero dans une collone ou une ligne avec des -1 partout dans cette même collone ou ligne. Dans ce cas, on est obliger de selectionner ce 0, c'est pour cela que l'on utilise la variable taken_x et taken_y.