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

Étant 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 symétrique, 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 calcule 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  }
```

À 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 intéressante que les villes précédemment étudier, si oui, on la mémorise puis on étudie 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 colonne 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 aux lignes 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 pénalité

Pour effectuer cela nous allons parcourir toute la matrice, et à chaque fois qu'on tombe sur un 0, on va calculer la pénalité de ce 0. La pénalité d'un 0 est la somme des plus petites valeurs de la ligne et de la colonne du 0. Il faut penser à exclure le 0 de la recherche. À chaque fois qu'on a calculé une pénalité, on la compare à la pénalité précédente, si elle est plus grande, on la mémorise.

```
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 zéro dans

une colonne ou une ligne avec des -1 partout dans cette même colonne ou ligne. Dans ce cas, on est obligé de sélectionner ce 0, c'est pour cela que l'on utilise la variable taken_x et taken_y et la valeur INFINITY.

```
    if(izero == -1 || jzero == -1)
    return;
```

Si on n'a pas trouvé de 0, on sort de la fonction. Les variables izero et jzero ont été initialisé à 1 avant de commencer la recherche.

2.3 Sélection de la liaison à effectuer

Une fois le 0 avec la plus grosse pénalité sélectionné, il faut maintenant mettre à jour la solution et la matrice des distances.

```
    starting_town[iteration] = izero;
    ending_town[iteration] = jzero;
```

Étant donné que l'on ne peut plus partir de la ville de départ de ou arriver à la ville d'arrivée de l'itération, on remplit la ligne et la colonne où se trouve le 0 par des -1.

De plus, étant donné que nous explorons le graphe en profondeur, il faut garder une version de la matrice au cas où le chemin ne mène pas à la solution finale. Pour cela, on utilise la matrice d2 comme matrice de travail et on conserve la matrice d.

```
    for (i = 0; i < NBR_TOWNS; i++)
    {
        d2[izero][i] = -1;
        d2[i][jzero] = -1;
    }
```

2.4 Élimination des circuits

Afin d'éviter à l'algorithme d'effectuer des itérations inutiles, il faut éliminer les circuits possibles dans la matrice des distances. Pour corriger cela, il faut parcourir la matrice des distances et mettre à -1 les chemins qui sont impossibles. Pour cela j'ai codé la fonction suivante :

```
void subtour_elimination(double d[NBR_TOWNS][NBR_TOWNS], int iteration)
{
    for(int i = 0; i <= iteration; i++)
    {
        int j = 0;
        int found = 1;
        int start = starting_town[i];
        int end = starting_town[i];

        while(j <= iteration && found)
        {
            for (int k = 0; k <= iteration; k++)
            {
                found = 0;
                if(starting_town[k] == end)
                {
                    end = ending_town[k];
                    found = 1;
                }
            }
            if(found)
            {
                d[end][start] = -1;
            }
            j++;
        }

    }
}
```

Pour effectuer cela, la fonction parcourt le vecteur des solutions, identifie les suites de chemins et élimine les possibilités qui feraient un circuit. Ainsi, si les solutions actuelles sont :

— 2 -> 3
— 4 -> 5
— 3 -> 1

On peut identifier les chemins suivants :

— 2 -> 3 -> 1
— 4 -> 5

Il faut donc éliminer les chemins 1 -> 2 et 5 -> 4. Pour cela, on parcourt la matrice des distances et on met à -1 les chemins qui sont impossibles.

2.5 Passage à l'itération suivante

2.5.1 On selectionne le 0 avec la plus grosse pénalité

Maintenant, c'est le moment de passer à l'itération suivante avec la ligne ci-dessous :

```
1     little_algorithm(d2, iteration + 1, eval_node_child);
```

De plus, dans le cas où nous sommes arrivé au bout, le nombre d'itérations est égal au nombre de villes, il faut donc construire la solution finale.

```
1     if (iteration == NBR_TOWNS)
2     {
3         build_solution ();
4         return;
5     }
```

2.5.2 On ne sélectionne pas le 0 avec la plus grosse pénalité

Dans le cas où le 0 avec la plus grosse pénalité n'est pas sélectionné, il faut reprendre la matrice d'origine et la mettre à jour la matrice des distances en mettant -1 au niveau du 0 pas sélectionné.

De plus, vu que nous n'avons pas sélectionné de chemin, il ne faut pas faire augment le nombre d'itérations.

```
1     /* Do the modification on a copy of the distance matrix */
2     memcpy (d2, d, NBR_TOWNS*NBR_TOWNS*sizeof(double)) ;
3
4     /**
5      *  Modify the dist matrix to explore the other possibility : the non-choice
6      *  of the zero with the max penalty
7      */
8
9     d2[izero][jzero] = -1;
10
11    /* Explore right child node according to non-choice */
12    little_algorithm(d2, iteration, eval_node_child);
```

2.6 Sortir de la fonction

2.6.1 On a trouvé une solution

Si on a trouvé une solution, il faut la comparer à la solution actuelle et la conserver si elle est meilleure. Ceci est effectué dans la fonction build_solution.

```
1     double eval = evaluation_solution(solution) ;
2
3     if (best_eval<0 || eval < best_eval)
4     {
5         best_eval = eval ;
6         for (i=0; i<NBR_TOWNS; i++)
7             best_solution[i] = solution[i] ;
```

```
        printf ("New best solution: ") ;
        print_solution (solution, best_eval) ;
    }
```

2.6.2 Le nœud actuel est moins intéressant que la meilleure solution

Dans le cas où le nœud actuel est moins intéressant que la meilleure solution, il faut sortir de la fonction avec les lignes suivantes :

```
if (best_eval>=0 && eval_node_child >= best_eval)
    return;
```

2.6.3 La matrice est pleine de -1

Si la matrice est pleine de -1, il faut sortir de la fonction. Ceci est possible, car en éliminant les cycles, on a potentiellement éliminé tous les chemins possibles.

```
if(izero == -1 || jzero == -1)
    return;
```

3 Expérimentation

On va maintenant tester nos algorithmes sur les données du Berlin 52. Pour cela on va caluler le temps d'exécution de l'algorithme de Little, de Little+ (élimination des sous tour) et de GLPK.

3.1 Résultats

Nb Villes	Little	Little+	GLPK
6	0,000776	0,008	0
11	2,875	0,026	0,1
16	>600	25,184	1,6
21	>600	315,251	27,9
26	>600	>600	>600
31	>600	>600	>600
36	>600	>600	>600
41	>600	>600	>600
46	>600	>600	>600
52	>600	>600	>600

On peut voir dans le tableau ci-dessus que l'algorithme de Little bien plus lent que Little+ (élimination des sous tour). Cela est dû au fait que l'algorithme de Little+ élimine les sous tours, ce qui permet d'éviter des itérations inutiles. Cependant, le problème étant un problème combinatoire, on se rend compte que malgré l'efficacité de l'algorithme de Little+ la durée de résonnement augmente de manière exponentielle. De plus on peut voir que GLPK est plus rapide que les deux autres algorithmes. J'en suis personnellement surpris, car GLPK est un solveur linéaire, alors que les deux autres algorithmes sont des algorithmes pensés spécifiquement pour résoudre ce problème.