IT45
TP3 - Algorithme de Little

Flavian Theurel

20 avril 2023

Table des matières

Le but de ce TP est d'implémenter l'algorithme de Little en C afin de résoudre le problème du voyageur de commerce (TSP). Voici le model mathématique du TSP :

Soit $x_{ij} = 1$ si l'arc entre la ville i et j est utilisé, 0 sinon

$$\begin{cases} minz = \sum_{ij} d_{ij}.x_{ij} & (1) \\ \forall i, \sum_{j} x_{ij} = 1 & (2) \\ \forall j, \sum_{i} x_{ij} = 1 & (3) \\ \{(i,j)/x_{ij} = 1\} : circuit & (4) \\ x_{ij} binary & (5) \end{cases}$$

1 Travail préparatoire

Pour compiler et exécuter le programme, j'ai utilisé le compilateur intégré au logiciel CLion, à savoir MinGW. Cela permet de compiler plus rapidement le programme que si l'on utilisait manuellement gcc en ligne de commande. Le programme affiche bien les coordonnées des 6 premières villes.

1.1 Compilation et exécution

Pour calculer la distance entre deux villes, il faut utiliser la formule suivante :

$$dist_{i \to j} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \quad (6)$$

On obtiendra une distance entre la ville 1 et 2 de $dist_{1 \to 2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Ainsi, le code permettant de calculer la matrice de distance est :

```
1    for (int i = 0; i < NBR_TOWNS; ++i) {
2        for (int j = 0; j < NBR_TOWNS; ++j) {
3            if (i == j) {
4                dist[i][j] = -1;
5            } else {
6                dist[i][j] = sqrt(pow((coord[j][0] - coord[i][0]),
     2) + pow((coord[j][1] - coord[i][1]), 2));
7            }
8        }
9    }
```

1.2 Fonction build_nearest_neighbour

Cette fonction doit permettre de générer un circuit en suivant l'heuristique du "plus proche voisin" et de retourner une évaluation de cette solution (la distance du parcours correspondant). Cette heuristique consiste à partir d'une ville donnée i et de se diriger vers la ville non visitée la plus proche.

Dans un premier temps, on suppose que la ville de départ est la 1. On initialise la solution ligne 9, puis l'on sélectionne la ville la plus proche en veillant à réinitialiser la variable min_dist à $+\infty$.
De la ligne 15 à 18, on vérifie que la ville i ne fait pas encore partie de la solution. La ligne 19 permet de tester si la ville i est la plus proche de la précédente. Si c'est le cas on actualise la valeur minimale de distance et la ville à l'étape correspondante de la solution.

```
1     double build_nearest_neighbour() {
2     /* solution of the nearest neighbour */
3     int sol[NBR_TOWNS];
4
5     /* evaluation of the solution */
6     double eval = 0;
7     int isUsed;
8
9     sol[0] = 0;
10    /*  Build an heuristic solution : the nearest neighbour */
11    for (int i = 1; i < NBR_TOWNS; ++i) { // Loop to select the N-1
       other cities' rank in the solution
12        double min_dist = INF; // Set the minimum distance to +inf
13        for (int j = 0; j < NBR_TOWNS; ++j) { // Find the nearest
      city for each step
14            isUsed = 0; // Check if the city is already used
15            for (int k = 0; k < i; ++k) {
16                if (sol[k] == j) isUsed = 1;
17            }
18            if (!isUsed) { // If not -> check if the city is the
      nearest
19                if (dist[sol[i - 1]][j] >= 0 && dist[sol[i - 1]][j]
       < min_dist) {
20                    min_dist = dist[sol[i - 1]][j];
21                    sol[i] = j;
22                }
23            }
24        }
25    }
26
27    eval = evaluation_solution(sol);
28    printf("Nearest neighbour ");
29    print_solution(sol, eval);
30    for (int i = 0; i < NBR_TOWNS; ++i) best_solution[i] = sol[i];
31    best_eval = eval;
32
33    return eval;
34 }
```

2 Algorithme de Little

2.1 Faire apparaître un zéro par ligne et par colonne

La première étape de l'algorithme de little consiste en la soustraction du minimum de chaque ligne et colonne à tous les éléments de ces dernières. Voici un exemple pour illustrer ces propos : $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 3 \\ 3 & 2 & 7 \end{pmatrix}$

Ici, il est nécessaire de soustraire 1 à la première ligne, 2 à la seconde et

à la troisième. On obtient donc $\begin{pmatrix} 0 & 1 & 2 \\ 0 & 3 & 1 \\ 1 & 0 & 5 \end{pmatrix}$ Il reste donc à soustraire 1 à la troisième colonne.

C'est ce que fait le code ci-dessous :

```
1   /* Check that there is at least one 0 in each row */
2   for (int i = 0; i < NBR_TOWNS; ++i) {
3       double min = INF;
4       for (int j = 0; j < NBR_TOWNS; ++j) {
5           if (d[i][j] < min) {
6               if (d[i][j] != -1)
7                   min = d[i][j];
8           }
9       }
10      for (int j = 0; j < NBR_TOWNS; ++j) {
11          if (d[i][j] != -1)
12              d[i][j] -= min;
13      }
14      if (min != INF)
15          eval_node_child += min;
16  }
17
18  /* Check that there is at least one 0 in each column */
19  for (int j = 0; j < NBR_TOWNS; ++j) {
20      double min = INFINITY;
21      for (int i = 0; i < NBR_TOWNS; ++i) {
22          if (d[i][j] < min) {
23              if (d[i][j] != -1)
24                  min = d[i][j];
25          }
26      }
27      for (int i = 0; i < NBR_TOWNS; ++i) {
28          if (d[i][j] != -1)
29              d[i][j] -= min;
30      }
31      if (min != INFINITY)
32          eval_node_child += min;
33  }
```

A la ligne 15 et 32, on actualise la valeur de la borne minimum du noeud actuel en lui ajoutant ce que l'on à précédemment retranché à la matrice.

2.2 Calcul des pénalités pour les zéros de la matrice

Pour trouver le 0 avec la plus forte pénalité, il est nécessaire de parcourir la matrice. Si la distance entre deux villes i et j est égale à 0, alors on calcul la pénalité correspondante à ce 0.
On suppose les valeurs des pénalités correspondant aux lignes et colonnes de la matrice égales à $+\infty$. De cette manière si les valeurs de la ligne et/ou de la colonne observé sont toutes égales à -1, c'est à dire que l'on est obligé de choisir soit la ville de départ soit celle d'arrivé, la pénalité totale sera automatiquement fixée à $+\infty$.
On recherche ensuite le minimum de la ligne et de la colonne avant d'additioner les pénalités. Le variables selected_x et selected_y (ligne 15) permettent de retenir si la valeure de la pénalité suivant x ou y à été modifiée. De cette manière, si aucune d'entre elles n'a été modifié, max_penalty est fixée à $+\infty$. Par opposi-

tion, lorque l'une des valeurs a été modifiée, max_penalty reçoit la valeur totale des pénalités.

```
    /* row and column of the zero with the max penalty */
    int izero = -1, jzero = -1;
    double max_penalty = -1;

    /**
     *  Store the row and column of the zero with max penalty in
     *  starting_town and ending_town
     */

    for (int i = 0; i < NBR_TOWNS; ++i) {
        for (int j = 0; j < NBR_TOWNS; ++j) {
            /* Check penalty only if it's a 0 */
            if (d[i][j] == 0) {
                double penalty_x = INF, penalty_y = INF;
                double selected_x = 0, selected_y = 0;
                /* Compute the penalty for a zero */
                for (int k = 0; k < NBR_TOWNS; ++k) {
                    if (k != i && d[k][j] < penalty_x && d[k][j] >=
     0) {
                        penalty_x = d[k][j];
                        selected_x = 1;
                    }
                    if (k != j && d[i][k] < penalty_y && d[i][k] >=
     0) {
                        penalty_y = d[i][k];
                        selected_y = 1;
                    }
                }

                /* Compute the total penalty of the current 0 */
                double penalty_total = penalty_x + penalty_y;

                /* Update the values */
                if (penalty_total > max_penalty && (selected_x ||
    selected_y)) {
                    max_penalty = penalty_total;
                    izero = i;
                    jzero = j;
                } else if (!selected_x && !selected_y) {
                    max_penalty = INF;
                    izero = i;
                    jzero = j;
                }
            }
        }
    }
```

2.3 Mise à jour du nouveau trajet à effecter

On ajoute le trajet correspondant au 0 possédant la plus grande pénalité aux tableaux starting_town et ending_town (ligne 7 et 8). Si aucun 0 n'est éligible, on arrête l'itération actuelle. (Il est possible de n'avoir aucun 0 éligible lorsque l'on utilise la fonction de suppression de sous-tours.)

```
    /* End if there is no 0 eligible */
    if (izero == -1 || jzero == -1)
        return;

```

```
    /* Save the starting and ending town */
    starting_town[iteration] = izero;
    ending_town[iteration] = jzero;
```

Comme on a choisi de se rendre d'une ville i à une ville j, il est maintenant impossible de repartir de la ville i, mais également d'arriver à la ville j une nouvelle fois (contraintes (2) et (3) du model mathématique). On doit donc mettre à -1 la ligne et la colonne correspondant au 0 sélectionné.

```
    for (int i = 0; i < NBR_TOWNS; ++i) {
        d2[starting_town[iteration]][i] = -1;
        d2[i][ending_town[iteration]] = -1;
    }
```

2.4 Evaluation de la solution

2.4.1 Evalutaion de la solution actuelle

Lorsque le nombre d'itération atteint le nombre de ville du TSP, on reconstruit la solution à partir des tableaux starting_town et ending_town et de la fonction build_solution.

Une fois la solution reconstruite, il faut la comparer avec la solution actuelle.

```
    double eval = evaluation_solution(solution);

    if (best_eval < 0 || eval < best_eval) {
        best_eval = eval;
        for (i = 0; i < NBR_TOWNS; ++i)
            best_solution[i] = solution[i];
        printf("New best solution: ");
        print_solution(solution, best_eval);
    }
```

2.4.2 La noeud actuel est moins intéressant

Si la nouvelle solution est meilleure, on la conserve, sinon on garde l'ancienne. De plus, si la valeur du noeud courant est supérieure ou égale à la longueur de la meilleure solution trouvée à l'intant t, on arrête l'itération actuelle.

```
    if (best_eval >= 0 && eval_node_child >= best_eval)
        return;t_solution(solution, best_eval);
    }
```

2.5 Passage à l'itération suivante

2.5.1 Le zéro avec la plus forte pénalité est sélectionné

On impose le choix correspondant au 0 à la plus forte pénalité. On exlpore donc le noeud suivant avec la matrice précédemment uitlisée.

```
    /* Explore left child node according to given choice */
    little_algorithm(d2, iteration + 1, eval_node_child);
```

La solution n'est construite que lorsque le nombre d'itération est égal au nombre de villes du problème.

2.5.2 Le zéro avec la plus forte pénalité n'est pas sélectionné

Dans ce cas, on impose le non-choix de l'arc allant de la ville i à la j. Il est donc nécessaire de repartir de la matrice initiale (ligne) tout en veillant à assigner -1 à $dist_{ij}$. De plus, comme le couple (i,j) n'a pas été retenu, il ne faut pas incrémenter le nombre d'itérations.

```
    /* Do the modification on a copy of the distance matrix */
    memcpy(d2, d, NBR_TOWNS * NBR_TOWNS * sizeof(double));

    /**
     *  Modify the dist matrix to explore the other possibility :
    the non-choice
     *  of the zero with the max penalty
     */
    d2[starting_town[iteration]][ending_town[iteration]] = -1;

    /* Explore right child node according to non-choice */
    little_algorithm(d2, iteration, eval_node_child);
```

2.6 Algorithme de Little+

L'algorithme de Little+ est une version optimisée de l'algorithme de Little. On remarque l'ajout d'une fonction permettant de supprimer les sous-tours avant que l'on ne les explore.
Le but de cet algorithme est d'accélérer l'algorithme initial.

2.6.1 Fonction de suppression des sous-tours

Dans le but d'éviter des calculs inutils, il est important de supprimer les sous-tours. En effet, la présence de ces derniers rend la solution inadéquate et ralonge le temps d'exécution global du programme.
Pour solutionner ce problème, Il faut parcourir la matrice de distance en remplaçant les arcs problèmatiques par des -1. Voici le code permettant d'y parvenir :

```
    void subtour_elimination(double matrix[NBR_TOWNS][NBR_TOWNS],
    int iteration) {
    for (int i = 0; i <= iteration; ++i) {
        /*
         * Initialize the circuit starting and ending points
         * We suppose that in the initial state a circuit is
    composed of a single city
         */
        int start_circuit = starting_town[i];
        int end_circuit = starting_town[i];

        /* Is a sub-tour found ? */
        int travel_found = 1;
        int iter = 0;

        /*
         * Check if there is at least 1 sub-tour starting in a
    specific city
         * while : allows to complete the circuit
         */
        while (iter <= iteration && travel_found) {
            for (int j = 0; j <= iteration; ++j) {
```

```
20                  travel_found = 0; // We suppose that the city is
    the last one of the circuit
21                  if (starting_town[j] == end_circuit) { // If the
    city is in the middle of the circuit
22                      travel_found = 1;
23                      end_circuit = ending_town[j];
24                  }
25              }
26
27              if (travel_found) {
28                  matrix[end_circuit][start_circuit] = -1;
29              }
30              iter++;
31          }
32      }
33  }
```

A la ligne 7 et 8, on initialise deux variables qui permettront de reformer des circuits. Il est bon de noter qu'à chaque itération, la ville initiale du circuit est différente.
La variable travel_found permet de déterminer si la ville testée est la dernière du circuit.
Ensuite, on construit le circuit en bouclant $i \in [0, iteration]$ fois sur les arcs de trajet.
Si un circuit à été trouvé, on bloque les valeurs inacceptables. Supposons que les arcs soient : $1 \to 3$, $2 \to 4$, $6 \to 2$, $4 \to 5$, $5 \to 5$, $3 \to 1$. On obtient les circuits : $1 \to 3$ et $2 \to 4 \to 5 \to 6$.
Il faut donc bloquer $3 \to 1$ et $6 \to 2$.

3 Expérimentation

Dans cette section, on testera l'efficacité des algorithmes de Little, de Little+ ainsi que le solveur GLPK.
Pour réaliser les mesures, on utilisera la fonction clock() de la librairie "time.h".
La durée d'exécution est calculée par la formule : $execution\_time = \frac{end - start}{CLOCKS\_PER\_SEC}$ avec $CLOCKS\_PER\_SEC$ le nombre de tick de l'horloge en une seconde.

3.1 Comparaison

Temps d'exécution en secondes			
Nombre de villes	Little	Little+	GLPK
6	0,047	0,036	0
10	0,289	0,053	0
15	171,266	3,449	0
20	600+	38,723	0,1
25	600+	600+	0,1
30	600+	600+	0,9
35	600+	600+	8,7
41	600+	600+	21,7
45	600+	600+	7,4
50	600+	600+	24,9

Le temps d'exécution étant, dans certains cas, extrêment long, j'ai décidé de tronquer les durées supérieures à 10 minutes. A titre indicatif, l'algorithme Little+ à 25 villes s'est exécuté en 2434,385 seconds soit une quarantaine de minutes.

On peut remarquer que l'algorithme de Little et Little+ ont tous deux une durée de raisonnement qui augmente de manière exponentielle. Il est toutefois bon de noter que l'algorithme de Little+, grâce à la suppression des sous-tours, est biens plus performant que son analogue. En effet, il ne dépasse la barre des 10 minutes que lorsqu'il traite du TSP à 25 villes ou plus. Little, lui, dépasse cette limite à partir de 20 villes.
Le solveur GLPK est sans appel le plus rapide. Il lui faut uniquement 24,9 secondes pour résoudre le TSP à 52 villes. Cependant, il manque cruellement de justesse au niveau des solutions qu'il produit. En effet, à cause de nombreuses incertitudes dans le calcul des valeurs, ce dernier produit des solutions imprécises.