IT45: TP3

Yann Derré

April 2023

Abstract

Le code est exécuté sur un processeur M1 et compilé avec le flag `-O3` pour bénéficier de l'accélération matérielle de la vectorisation des calculs. Code source est disponible sur mon Repo GitHub.

Contents

1 Travail préparatoire

1.1 Calcul des distances

Pour utiliser l'algorithme de Little, nous calculons la matrice de distances entre une ville A et B. Nous utilisons la formule de la distance entre 2 points dans une space cartésien à 2 dimensions.

$$D_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} avec \forall i \neq j$$

Voici la fonction en C:

```
1  void compute_dist_matrix()
2  {
3      for (int i = 0; i < NBR_TOWNS; i++)
4      {
5          int j = 0;
6          while (j < i)
7          {
8              dist[i][j] = sqrt(pow(coord[i][0] - coord[j][0], 2)
9              + pow(coord[i][1] - coord[j][1], 2));
10             dist[j][i] = dist[i][j];
11             j++;
12         }
13         dist[i][i] = -1;
14     }
15 }
```

A la ligne 12, nous appliquons dans la matrice de la distance une restriction du trajet la ville sur elle-même en lui donnant la valeur -1.

1.2 Création de l'heuristique "nearest neighbour"

Pour faire fonctionner l'algorithme de Little, nous utilisons comme heuristique la borne initiale de la distance parcourue pour ignorer les branches non optimales. Cela nous donne également une 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      // On initialise les villes à 0
10     for (i = 0; i < NBR_TOWNS; i++)
11     {
12         sol[i] = 0;
13     }
14
15     //Trouve la ville la plus proche de la ville actuelle
16     for (i = 1; i < NBR_TOWNS; i++)
17     {
18         double min = -1;
19         int k;
20         for (int j = 0; j < NBR_TOWNS; j++)
21         {
22             for (k = 0; k < i; k++)
23             {
```

```
                if (sol[k] == j)
                {
                    break;
                }
            }
            if (k < i)
            {
                //La ville est déjà visitée?
                continue;
            }
            if (min < 0 || dist[sol[i - 1]][j] < min)
            {
                //Ville actuelle plus intéressante
                min = dist[sol[i - 1]][j];
                sol[i] = j;
            }
        }
    }

    eval = evaluation_solution(sol);
    printf("Nearest neighbour ");
    print_solution(sol, eval);

    for (i = 0; i < NBR_TOWNS; i++)
        best_solution[i] = sol[i];
    best_eval = eval;

    return eval;
}
```

2 Algorithme de Little

2.1 Setup matrice des distances

On fait apparaître un zéro dans chaque colonne de la matrice des distances en effectuant la soustraction avec le minimum de chaque colonne.

```
    for (i = 0; i < NBR_TOWNS; i++)
    {
        double min = INFINITY;
        for (j = 0; j < NBR_TOWNS; j++)
        {
            if (d[i][j] < min)
            {
                if (d[i][j] != -1)
                    min = d[i][j];
            }
        }
        for (j = 0; j < NBR_TOWNS; j++)
        {
            if (d[i][j] != -1)
                d[i][j] -= min;
        }
        if (min != INFINITY)
            eval_node_child += min;
    }
```

```
    // Check if 0 in every column

    for (j = 0; j < NBR_TOWNS; j++)
    {
        double min = INFINITY;
        for (i = 0; i < NBR_TOWNS; i++)
        {
            if (d[i][j] < min)
            {
                if (d[i][j] != -1)
                    min = d[i][j];
            }
        }
        for (i = 0; i < NBR_TOWNS; i++)
        {
            if (d[i][j] != -1)
                d[i][j] -= min;
        }
        if (min != INFINITY)
            eval_node_child += min;
    }
```

La distance -1 est également ignorée pour ne pas explorer les villes marquées dans la création de la matrice des distances. On vérifie que la nouvelle valeur trouvée par eval ne dépasse pas la meilleure trouvée, en stoppant la recherche du node si le check ne passe pas

```
    /* Cut : stop the exploration of this node */
    if (best_eval >= 0 && eval_node_child >= best_eval)
        return;
```

2.2 Identifier les pénalités

On identifie le zéro avec la plus grosse pénalité, c'est-à-dire la somme des plus petites valeurs de la ligne et de la colonne 0. On itère sur chaque 0 en comparant avec la pénalité précédente, en fixant une pénalité infinie sur les 0 déjà explorés et mémorisant la pénalité la plus grande.

```
    int izero = -1, jzero = -1;
    double max_penalty = -1;
    for (i = 0; i < NBR_TOWNS; i++)
    {
        for (j = 0; j < NBR_TOWNS; j++)
        {
            if (d[i][j] == 0)
            {
                double penalty_x = INFINITY, penalty_y = INFINITY;
                double visited_x = 0, visited_y = 0;
                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];
                        visited_x = 1;
                    }
                    if (k != j && d[i][k] < penalty_y && d[i][k] >= 0)
                    {
                        penalty_y = d[i][k];
                        visited_y = 1;
                    }
```

```
                }
                double penalty = penalty_x + penalty_y;
                if (penalty > max_penalty && (visited_x || visited_y))
                {
                    max_penalty = penalty;
                    izero = i;
                    jzero = j;
                }
                else if (visited_x == 0 && visited_y == 0)
                {
                    max_penalty = INFINITY;
                    izero = i;
                    jzero = j;
                }
            }
        }
    }

//Si on ne trouve pas de 0, sortie !! Pour plus tard, l'élimination des sous-tours peut avoir supprimé l
    if (izero == -1 || jzero == -1)
    {
        return;
    }
//Mise à jour de la solution avec notre 0 choisi.
    starting_town[iteration] = izero;
    ending_town[iteration] = jzero;

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

    /* Modify the dist matrix to explore the choice of the zero with the max penalty */
    for (i = 0; i < NBR_TOWNS; i++)
    {
        d2[izero][i] = -1;
        d2[i][jzero] = -1;
    }
```

On travaille sur une copie D2 de la matrice des distances, car nous utilisons un algo DFS pour l'exploration du graphe. La ville de départ et d'arrivée sont également bannies, assignant -1 dans la matrice.

La transition à l'itération suivante se fait en explorant le 0 choisi :

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

    if (iteration == NBR_TOWNS)
    {
        build_solution();
        return;
    }
```

Nous construisons la solution finale si une solution contenant le nombre de villes est trouvée.

Sinon, on reprend notre copie "d" originale de la matrice dans "d2" et nous éliminons le 0 exploré en le bannissant avec -1.

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

    d2[izero][jzero] = -1;
```

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

2.3 Comparaison solution

La fonction *build_ solution* se charge d'évaluer la solution trouvée à la meilleure solution :

```
    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);
    }
    return;
}
```

3 Expérimentation

3.1 Little+ élimination des sous-tours

Pour optimiser la recherche et éviter les itérations inutiles, on élimine les potentielles boucles dans la matrice

```
void remove_loops(double d0[NBR_TOWNS][NBR_TOWNS], int iteration)
{
    int start_town;
    int end_town;
    int j, k;
    bool breakout;
    for (int i = 0; i <= iteration; i++)
    {
        j = 0;
        start_town = starting_town[i];
        end_town = starting_town[i];
        breakout = true;
        while (j <= iteration && breakout)
        {
            j++;
            for (k = 0; k <= iteration; k++)
            {
                breakout = false;
                if (end_town == starting_town[k])
                {
                    end_town = ending_town[k];
                    breakout = true;
                }
            }
            if (breakout)
            {
                d0[end_town][start_town] = -1;
            }
        }
```

```
30      }
31  }
```

Dans les chemins possibles *1->2 // 2->3 // 4->5*, on peut se retrouver avec les 2 circuits 1->2->3->1 et 4->5->4. Cette fonction va bannir les transitions 3->1 et 5->4 ou plus génériquement **end_town -> start_town** pour éviter cela en marquant -1 dans la matrice des distances.

3.2 Benchmark Little/Little+/Modèle GMPL

3.3 Mesure des performances

Avec Little:

- Pour 6 villes, la solution optimale trouvée : 0 1 2 3 5 4, avec une distance de 2315.15
- Pour 11 villes, on obtient 0 1 6 2 7 8 9 10 3 5 4 avec une distance de 4038.44

L'algorithme de Little n'arrive cependant pas à trouver une solution en moins de 10 min pour 16 villes

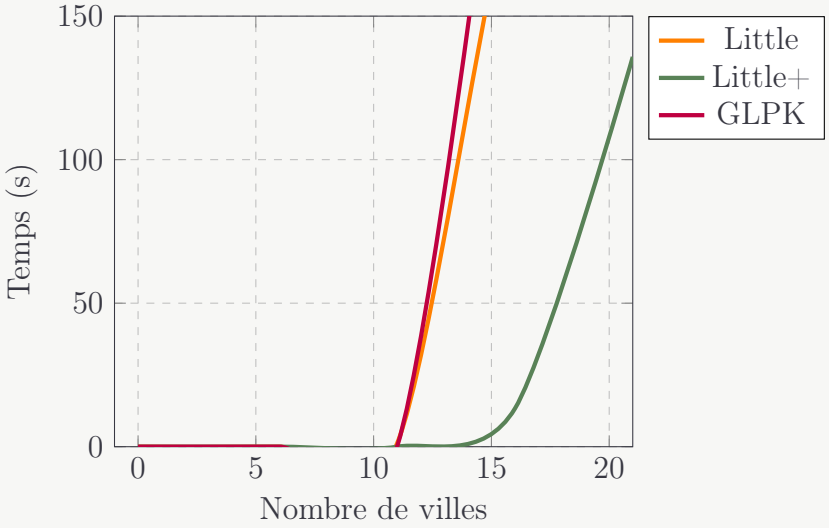
Avec Little+:

- on trouve pour 16 villes 0 15 13 12 10 11 3 5 4 14 9 8 7 2 6 1 avec une distance de 4990.46 en seulement 13s.
- on trouve pour 21 villes 0 17 20 1 6 16 2 18 7 8 9 14 4 5 3 11 10 12 13 15 19 avec une distance de 5291 en 136s.

GLPK nous permet de trouver une solution à 16 villes en 1.3s mais le temps de résolution devient exponentiel au-delà de 20 villes.

Villes	Little	Little+	GLPK
6	0.000386	0.000167	0
11	1.226015	0.005285	0.1
16	>300	13.312551	1.1
21	>300	135.795077	268.2
21+	>300	>300	>300

Table 1: Comparaison timings entre les implémentations du TSP en secondes



Au-delà de 20 villes, le temps de calcul devient prohibitive pour les 3 implémentations par sa nature exponentielle.

Little+ possède la meilleure implémentation, en retirant les sous-tours, on économise des itérations et du temps de recherche.