IT45 - Optimisation et recherche opérationnelle

BENEDUCI Marie

18/04/2023

Table des matières

1. Introduction 3
2. Travail préparatoire 4
    1. Compilation et exécution 4
    2. Calcul et affichage de la matrice des distances 4
    3. Construction d'une solution suivant l'heuristique « plus proche voisin » 5
3. Algorithme de Little 7
4. Expérimentation 12
    1. Test sur jeux de données de dimension faible (6 et 10 villes) 12
    2. Evolution du temps de recherche quand la dimension du problème augmente 13
    3. Méthode de Little et solveur GLPK 15
    4. Optimisation de l'algorithme Little C 17
5. Conclusion 20

1. Introduction

Ce document est un rapport d'analyse sur un TP réalisé en IT45 à l'UTBM. L'objectif de ce dernier est d'implémenter l'algorithme de Little en C pour résoudre le problème du voyageur de commerce (TSP) et de comparer les performances de différentes méthodes d'optimisation du TSP (solveur/Little principalement) au travers de plusieurs exercices guidées.

Vous trouverez dans ce rapport une explication et une analyse de chaque étape de programmation suivant le TP, suivi d'une description, explication et comparaison des différents programmes.

Le code source du projet ainsi que de ce rapport sont disponibles sur mon Github :

https://github.com/Mxrie2001/IT45-Little

2. Travail préparatoire

1. Compilation et exécution

Dans ce TP, ayant le choix par rapport à la compilation et l'exécution du programme, j'ai choisi de les faire avec les commandes gcc suivantes :

Dans un premier temps il faut créer le fichier .o à partir du .c

```
gcc -c little.c -o little.o
```

Puis créer un fichier exécutable en autorisant toutes les librairies

```
gcc little.o -o app -
lm
```

Et enfin, lancer le programme

```
./app
```

2. Calcul et affichage de la matrice des distances

Pour réaliser cette fonction c en se basant sur les calculs de l'énoncé ci-dessous :

On posera : dist[i][i] = -1. Dans le programme, les coordonnées de la ville i sont : $x_i = coord[i][0]$ et $y_i = coord[i][1]$. Par conséquent la distance de la ville i à la ville j est : $dist[i][j] = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$.

Nous arriverons au code suivant qui nous permet de déterminer la distance entre 2 villes :

```
for (i=0; i<NBR_TOWNS; i++)
    {
        for (j=0; j<NBR_TOWNS; j++)
        {
            if (i==j)
                dist[i][j]=-1;
            else
            {
                dist[i][j] = sqrt(pow(coord[j][0]-coord[i][0],2) + pow(coord[j][1]-coord[i][1],2));
            }
        }
    }

    printf ("Distance Matrix:\n") ;
    print_matrix (dist) ;
    printf ("\n") ;
```

Les fonctions pow() et sqrt() étant dans la librairie « math.h », il faut donc l'inclure à notre programme avec `#include <math.h>`. Ces dernières permettent respectivement de mettre à une certaine puissance (ici 2) des données et d'en faire la racine carrée.

3. Construction d'une solution suivant l'heuristique « plus proche voisin »

En se basant sur le code et les fonctions déjà existante et en décommentant le code utile, il nous faut, pour réaliser l'heuristique « plus proche voisin » :

- Isoler les villes 2 à 2, la ville de départ(courante) et la prochaine ville
- Vérifier que la ville de destination n'est pas déjà dans le tableau de solution
- Si elle n'est pas dans le tableau de solution et qu'elle est différente de la ville courante, alors si la distance entre ces dernières est inférieure à la distance minimale, la ville de destination entre dans la solution comme prochaine ville
- Sa distance avec la dernière est garder en mémoire comme minimum de distance
- La boucle est répétée jusqu'à ce qu'il n'y ai plus de ville à comparer
- Puis la solution est évalué par la fonction existante evaluation_solution()
- La fonction de construction d'une solution suivant l'heuristique « plus proche voisin » fini par retenir la meilleure solution grâces aux fonctions déjà existantes et appelées ici.
- La fonction renvoie son évaluation.

N.B. Grace aux « print » nous pouvons avoir un aperçu dans la console des résultats et vérifier que notre fonction fonctionne bien.

```
Nearest neighbour (3278.84): 0 2 7 9 8 4 5 3 6 1
New best solution: (2968.01): 0 4 3 5 8 9 7 2 1 6
New best solution: (2913.95): 0 4 5 3 8 9 7 2 1 6
New best solution: (2826.50): 0 1 6 2 7 8 9 3 5 4
Best solution:(2826.50): 0 1 6 2 7 8 9 3 5 4
```

Au niveau du code, nous arrivons donc à cela :

```
double build_nearest_neighbour()
{
    /* solution of the nearest neighbour */
    int i, sol[NBR_TOWNS] ;

    /* evaluation of the solution */
    double eval = 0 ;

    sol[0] = 0 ;

    /**
     *  Build an heuristic solution : the nearest neighbour
    */

    int ville;
        int nextville = 0;
        double min;

        for (int k=1; k<NBR_TOWNS; k++)
        {
            ville = nextville;
            min = 999999;
            for (int j=0; j<NBR_TOWNS-1; j++)
            {
                bool villeDejaDansSol = valInTab(j+1, sol);
                //si la ville d'apres n'est pas déjà dans le tableau de solution et si la prochaine ville
n'est pas celle que l'on compare actuellement
                if (!villeDejaDansSol && j+1 != ville)
                {
                    //si la distance entre les ville est plus petite que le minimum
                    if (dist[ville][j+1] < min)
                    {
                        //nouvelle prochaine ville
                        nextville = j+1;
                        //nouveau minumum
                        min = dist[ville][j+1];
                    }
                }
                else
                {
                }
            }
            //La nouvelle ville suivante s'ajoute au tableau des solutions
            sol[k]=nextville;
        }

    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 ;
}
```

3. Algorithme de Little

L'algorithme fonctionne en explorant toutes les permutations possibles des villes à visiter et en sélectionnant la permutation qui minimise la distance totale parcourue.

La fonction little_algorithm() est la fonction récursive principale qui effectue la recherche de toutes les permutations possibles. La fonction prend en entrée une matrice de distances entre les villes, l'itération actuelle de l'algorithme, et l'évaluation de la solution parente. L'algorithme s'arrête lorsque toutes les villes ont été visitées (c'est-à-dire lorsque l'itération est égale au nombre de villes) ou si la distance totale de la solution actuelle dépasse la meilleure distance connue à ce jour.

La fonction commence par effectuer une réduction de la matrice de distances en soustrayant le minimum de chaque ligne et colonne de la matrice. Ensuite, elle détermine le zéro ayant la plus grande pénalité (qui représente le choix optimal à faire pour la prochaine ville à visiter) et stocke les indices correspondants dans des tableaux. La fonction vérifie également si la solution actuelle est toujours prometteuse en comparant l'évaluation actuelle avec la meilleure évaluation connue à ce jour.

Si la fonction trouve qu'il n'y a pas de zéro dans la matrice, elle retourne sans rien faire. Si elle trouve un zéro avec la plus grande pénalité, elle crée deux copies de la matrice de distances et modifie l'une d'entre elles pour représenter le choix optimal, puis appelle récursivement la fonction little_algorithm() avec cette matrice modifiée en tant que matrice de distances, l'itération actuelle plus un, et l'évaluation de la solution actuelle plus le coût du choix optimal. Ensuite, elle modifie l'autre copie de la matrice de distances pour représenter le choix de ne pas choisir le zéro avec la plus grande pénalité, puis appelle récursivement la fonction little_algorithm() avec cette matrice modifiée en tant que matrice de distances, l'itération actuelle, et l'évaluation de la solution actuelle.

```c
void little_algorithm(double d0[NBR_TOWNS][NBR_TOWNS], int iteration, double eval_node_parent)
{

    //if (iteration == 0)
    //{
    //    printf("--- Iteration %d ---\n",iteration);
    //}

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


    /* Do the modification on a copy of the distance matrix */
    double d[NBR_TOWNS][NBR_TOWNS] ;
    memcpy (d, d0, NBR_TOWNS*NBR_TOWNS*sizeof(double)) ;

    //printf ("-- Start --\n") ;
    //print_matrix (d) ;
    //printf ("\n") ;

    int i, j ;

    double eval_node_child = eval_node_parent;

    /**
    * substract the min of the rows and the min of the columns
    * and update the evaluation of the current node
    */

    //On place un 0 sur chaque ligne et colonne de la matrice et on récupère la somme des minimums
enlevés
    //On met a jour l'évaluation du noeud "enfant" avec le total récupéré
    eval_node_child += reduct_matrice(d, 0);
    eval_node_child += reduct_matrice(d, 1);

    //if (iteration == 0)
    //{
    //    printf("borne mini: %f \n", eval_node_child) ;
    //}



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


    /**
    *  Compute the penalities to identify the zero with max penalty
    *  If no zero in the matrix, then return, solution infeasible
    */

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

    for (int i=0; i<NBR_TOWNS; i++)
    {
        for (int j=0; j<NBR_TOWNS; j++)
        {
            if (d[i][j]==0)
            {
                double pen = count_penalites(d, i, j);
                if (pen > maxpen)
                {
                    maxpen = pen;
                    izero = i;
                    jzero = j;
                }
            }
        }
    }
```

```
    //if (iteration == 0)
    //{
    //    printf("max penalité: %f \n", maxpen) ;
    //    printf("premier zéro ayant la pénalité la plus forte (départ): %d \n", izero) ;
    //    printf("premier zéro ayant la pénalité la plus forte (arrivée): %d \n", jzero) ;
    //}

    //printf ("-- Mid --\n") ;
    //print_matrix (d) ;
    //printf ("\n") ;


    /**
    *  Store the row and column of the zero with max penalty in
    *  starting_town and ending_town
    */
    starting_town[iteration] = izero;
    ending_town[iteration] = jzero;

    // Little+ Optimisation
    //if (jzero == 0 && izero == ending_town[iteration - 1]){
    //    return;
    //}



    if (maxpen == -1)
        return;

    /* 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 matrix d2 according to the choice of the zero with the max penalty
    */

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

    d2[jzero][izero] = -1;

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

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

    //printf ("-- End --\n") ;
    //print_matrix (d) ;
    //printf ("\n") ;

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

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

}
```

Pour la réalisation de cette fonction en suivant toutes les étapes de l'énoncé et du code donné :

- Dans un premier temps nous vérifions si le nombre d'itération correspond au nombre de ville à évaluer, si c'est le cas, nous construisons une solution, la fonction s'arrête. Sinon la fonction continue
- Ensuite nous créons une copie de la matrice des distances
- On place un 0 sur chaque ligne et colonne de la matrice et on récupère la somme des minimums enlevés grâce à la fonction de reduction de matrice qui utilise elle-même une fonction permettant d'avoir le minimum des ligne et colonne afin de le soustraire a la ligne/colonne correspondante.

```
double reduct_matrice(double d0[NBR_TOWNS][NBR_TOWNS], int type)
{
    double total = 0;
    for (int i = 0; i < NBR_TOWNS; i++)
    {
        //on trouve le minimum de la ligne ou colone et on le stocke dans une variable
        double min = minLigneCol(d0,i, type);

        if (min != 999999)
        {
            for (int j = 0; j < NBR_TOWNS; j++) //Pour chaque ligne/Colones on enleve le minimum aux
données
            {
                if (type == 0 && d0[i][j] != -1){
                    d0[i][j] = d0[i][j] - min;
                }
                else if (type == 1 && d0[j][i] != -1){
                    d0[j][i] = d0[j][i] - min;
                }

            }

            total += min; // on récupere le total des minimums
        }
    }
    return total;
}
```

```
double minLigneCol(double d0[NBR_TOWNS][NBR_TOWNS], int i, int colOuLigne)
{
    double min = 999999;
    for (int j = 0; j < NBR_TOWNS; j++)
    {
        //traitement des lignes
        if (colOuLigne == 0)
        {
            // Si la valeur vaut -1, c'est qu'on l'a déjà traité/ on est sur une diagonale
            if (d0[i][j] != -1 && d0[i][j] < min){
                min = d0[i][j];
            }
        }else{ //traitement des lignes
            if (d0[j][i] != -1 && d0[j][i] < min){
                min = d0[j][i];
            }
        }
    }

    return min;
}
```

Ici nous allons réduire la matrice à traiter en cherchant le minimum, vérifiant qu'il n'est pas dans la diagonale en temps que -1. Si le minimum =0 nous ne faisons rien. S'il est différent, on le soustrait a la ligne ou colonne de la matrice jusqu'à obtenir un 0 par ligne ou colonnes (en fonction de l'appel de la fonction).

- On vérifie si la solution est optimale
- On recherche le 0 avec la plus grande pénalité grâce à la fonction count_penalites()

```
double count_penalites(double d0[NBR_TOWNS][NBR_TOWNS], int x, int y)
{
    double min_row = 999999;
    double min_col = 999999;

    for (int t=0; t<NBR_TOWNS; t++)
    {
        // boucle sur les lignes qui n'ont pas été traitées
        if (y != t && d0[x][t] >= 0)
        {
            if (d0[x][t] < min_row)
                min_row = d0[x][t];
        }

        if (min_row == 999999)
            min_row = 0.0;

        // boucle sur les colonne qui n'ont pas été traitées
        if (x != t && d0[t][y] >= 0)
        {
            if (d0[t][y] < min_col)
                min_col = d0[t][y];
        }

        if (min_col == 999999)
            min_col = 0.0;
    }
    return min_col + min_row; // on récupere le total des minimums colonne et ligne pour la pénalité
}
```

On va chercher le minimum de la ligne et de la colonne du 0 en question en l'excluant et on additionne les résultats pour avoir la pénalité.

- Ensuite on stock la ligne et la colonne du 0 avec la pénalité maximale dans la ville de départ et d'arrivée
- On vérifie que le max des pénalités n'est pas -1
- On fait les modifications nécessaires sur la copie de la matrice des distances.
- On modifie ensuite la matrice des distances pour voir les possibilités alternatives.

N.B. cette fonction sera lancé en boucle pour avoir le résultat optimal de l'algorithme Little avec les n itérations nécessaires.

4. Expérimentation

1. Test sur jeux de données de dimension faible (6 et 10 villes)

<u>Résultat terminal :</u>

Pour Les 6 premières villes :

```
Points coordinates:
Node 0: x=565.000000, y=575.000000
Node 1: x=25.000000, y=185.000000
Node 2: x=345.000000, y=750.000000
Node 3: x=945.000000, y=685.000000
Node 4: x=845.000000, y=655.000000
Node 5: x=880.000000, y=660.000000

Distance Matrix:
0:  -1.0  666.1  281.1  395.6  291.2  326.3
1: 666.1   -1.0  649.3 1047.1  945.1  978.1
2: 281.1  649.3   -1.0  603.5  508.9  542.5
3: 395.6 1047.1  603.5   -1.0  104.4   69.6
4: 291.2  945.1  508.9  104.4   -1.0   35.4
5: 326.3  978.1  542.5   69.6   35.4   -1.0

Nearest neighbour (2608.26): 0 2 4 5 3 1
New best solution: (2324.69): 0 1 2 4 3 5
New best solution: (2323.20): 0 4 3 5 2 1
New best solution: (2315.15): 0 1 2 3 5 4
Best solution:(2315.15): 0 1 2 3 5 4


Execution time : 0.000225

Hit RETURN!
```

```
--- Iteration 0 ---
borne mini: 1754.406226
max penalité: 34.761124
premier zéro ayant la pénalité la plus forte (départ): 3
premier zéro ayant la pénalité la plus forte (arrivée): 5
```

Pour les 10 premières villes :

```
Points coordinates:
Node 0: x=565.000000, y=575.000000
Node 1: x=25.000000, y=185.000000
Node 2: x=345.000000, y=750.000000
Node 3: x=945.000000, y=685.000000
Node 4: x=845.000000, y=655.000000
Node 5: x=880.000000, y=660.000000
Node 6: x=25.000000, y=230.000000
Node 7: x=525.000000, y=1000.000000
Node 8: x=580.000000, y=1175.000000
Node 9: x=650.000000, y=1130.000000

Distance Matrix:
0:  -1.0  666.1  281.1  395.6  291.2  326.3  640.8  426.9  600.2  561.5
1: 666.1   -1.0  649.3 1047.1  945.1  978.1   45.0  956.2 1135.0 1133.0
2: 281.1  649.3   -1.0  603.5  508.9  542.5  610.6  308.1  485.6  487.3
3: 395.6 1047.1  603.5   -1.0  104.4   69.6 1026.4  525.0  611.0  533.9
4: 291.2  945.1  508.9  104.4   -1.0   35.4  923.6  470.6  583.6  513.5
5: 326.3  978.1  542.5   69.6   35.4   -1.0  957.0  491.6  596.0  523.3
6: 640.8   45.0  610.6 1026.4  923.6  957.0   -1.0  918.1 1095.9 1095.7
7: 426.9  956.2  308.1  525.0  470.6  491.6  918.1   -1.0  183.4  180.3
8: 600.2 1135.0  485.6  611.0  583.6  596.0 1095.9  183.4   -1.0   83.2
9: 561.5 1133.0  487.3  533.9  513.5  523.3 1095.7  180.3   83.2   -1.0

Nearest neighbour (3278.84): 0 2 7 9 8 4 5 3 6 1
New best solution: (2968.01): 0 4 3 5 8 9 7 2 1 6
New best solution: (2913.95): 0 4 5 3 8 9 7 2 1 6
New best solution: (2826.50): 0 1 6 2 7 8 9 3 5 4
Best solution:(2826.50): 0 1 6 2 7 8 9 3 5 4


Execution time : 0.003459

Hit RETURN!
```

```
--- Iteration 0 ---
borne mini: 1200.591572
max penalité: 933.786219
premier zéro ayant la pénalité la plus forte (départ): 1
premier zéro ayant la pénalité la plus forte (arrivée): 6
```

En comparant les résultats du terminal au résultat donnés dans l’énoncé, nous pouvons conclure que l’algorithme fonctionne bien, que ce soit au niveau de la recherche de solution optimale ou encore dans ses différentes étapes au niveau des itérations pour obtenir ici borne minimum, 1er zéro ayant la pénalité la plus forte pour le départ et l’arrivée.

2. Evolution du temps de recherche quand la dimension du problème augmente

Afin de comparer au mieux les résultats, et d'utiliser le fichier. tsp, une fonction de lecture de fichier avec remplissage de tableau à été rajouté au code :

```c
void lire_fichier()
{
    FILE* file = fopen("berlin52.tsp", "r");
    if(file == NULL)
    {
        perror("Error file opening");
        fflush(stdout);
        exit(EXIT_FAILURE);
    }

    char temp[100];
    while(strcmp(temp, "NODE_COORD_SECTION") != 0)
    {
        fscanf(file, "%s", temp);
    }

    //Boucle pour remplir le tableau avec les données du fichier
    for(int j = 0; j < NBR_TOWNS; j++)
    {
        int temp_nbr;
        fscanf(file, "%d", &temp_nbr);
        fscanf(file, "%f", &coord[j][0]);
        fscanf(file, "%f", &coord[j][1]);

        fgets(temp, 100, file);
    }
    fclose(file);
}
```

De même pour évaluer le temps d'exécution, la fonction clock() est utilisée.

```c
    //debut du chrono (pour evaluer le temps d'execution)
    clock_t start, end;
    double time_spent;
    start = clock();

    int iteration = 0 ;
    double lowerbound = 0.0 ;

    little_algorithm(dist, iteration, lowerbound) ;
    //fin du chrono (pour evaluer le temps d'execution)
    end = clock();

    printf("Best solution:") ;
    print_solution (best_solution, best_eval) ;

    time_spent = ((double) (end - start)) /
CLOCKS_PER_SEC;
    printf("\nExecution time : %f\n\n", time_spent);
```

Comparaison temps d'exécution Little en fonction du nombre de villes (pas de 5 villes en partant de 6) :

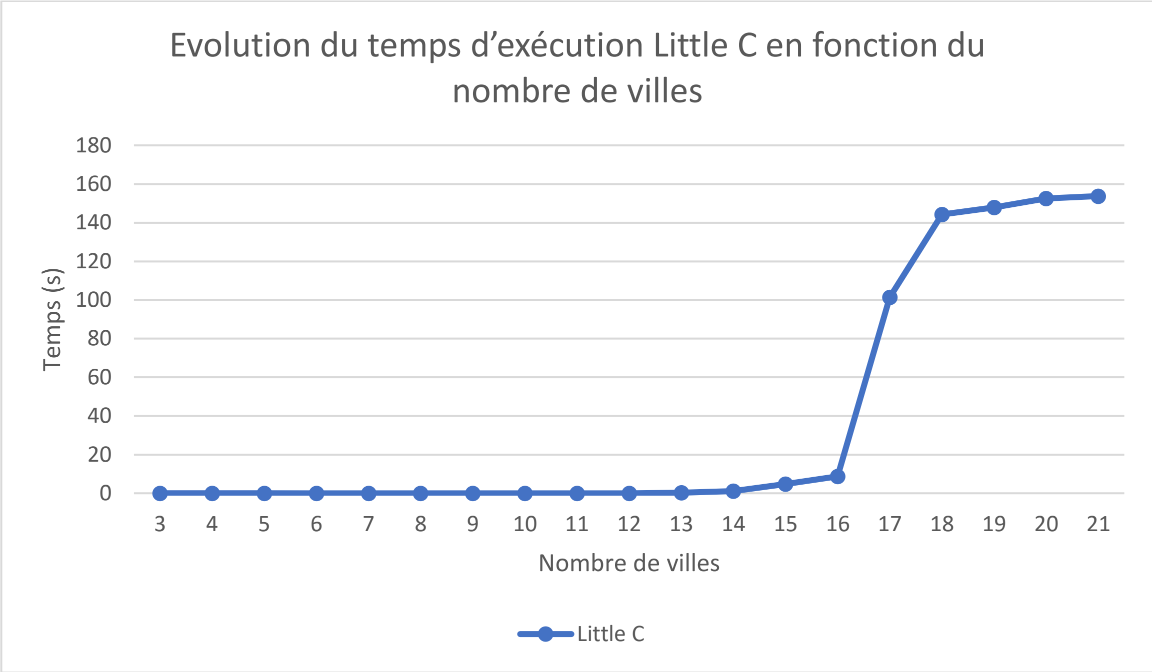
Nombre de villes	Temps d'exécution (en s)
6	0.000278
11	0.009417
16	8.750216
21	153.756221

Ici pour le test, nous nous arrêtons à 21 villes. Le temps d'exécution devient de plus en plus élevé, et augmente d'une manière exponentielle. Nous sommes déjà à presque 3min d'exécution pour les 21 villes).

Si nous regardons les choses plus précisément avec un pas de 1 villes en partant de 3 :

Comparaison temps d'exécution Little en fonction du nombre de villes :

Nombre de villes	Temps d'exécution (en s)
3	0.000005
4	0.000032
5	0.000051
6	0.000278
7	0.000543
8	0.001173
9	0.002145
10	0.007552
11	0.009417
12	0.042689
13	0.334048
14	1.137067
15	4.790570
16	8.750216
17	101.443994
18	144.185983
19	147.812311
20	152.443007
21	153.756221



Nous pouvons dire que l'algorithme est très efficace jusqu'à 16 villes, car ensuite son temps d'exécution dépasse la minute et augmente de manière exponentielle, puis il se restabilise aux alentours de 2-3 minutes d'exécutions pour 18-21 villes mais repart ensuite de manière exponentielle pour dépasse 1 heure d'exécution pour les 52 villes (données non présentes sur le graphique mais le programme avait dépassé 1h et n'avait pas terminé son exécution.). Nous pouvons en déduire que l'exécution se fait par mini pallier de 2-3 villes de différence, mais des ce dernier dépassé, le temps d'exécution augmente exponentiellement.

N.B. Pour les 52 villes, la solution optimale a une distance de 7544.37. Il faut plusieurs heures à Little C pour le compiler.

3. Méthode de Little et solveur GLPK

N.B. Le code GLPK n'étant pas demandé, je me suis permise de le prendre sur internet pour faire mes comparaisons et analyses. Il a été exécuté via le terminal avec la commande : 

Résultat GLPK pour les 6 premières villes :

```
INTEGER OPTIMAL SOLUTION FOUND
Time used:   0.0 secs
Memory used: 0.2 Mb (237304 bytes)
Display statement at line 26
(0,4)
(1,0)
(2,1)
(3,2)
(4,5)
(5,3)
Display statement at line 27
2315.14691286856
Model has been successfully processed
```

Résultat GLPK pour les 10 premières villes :

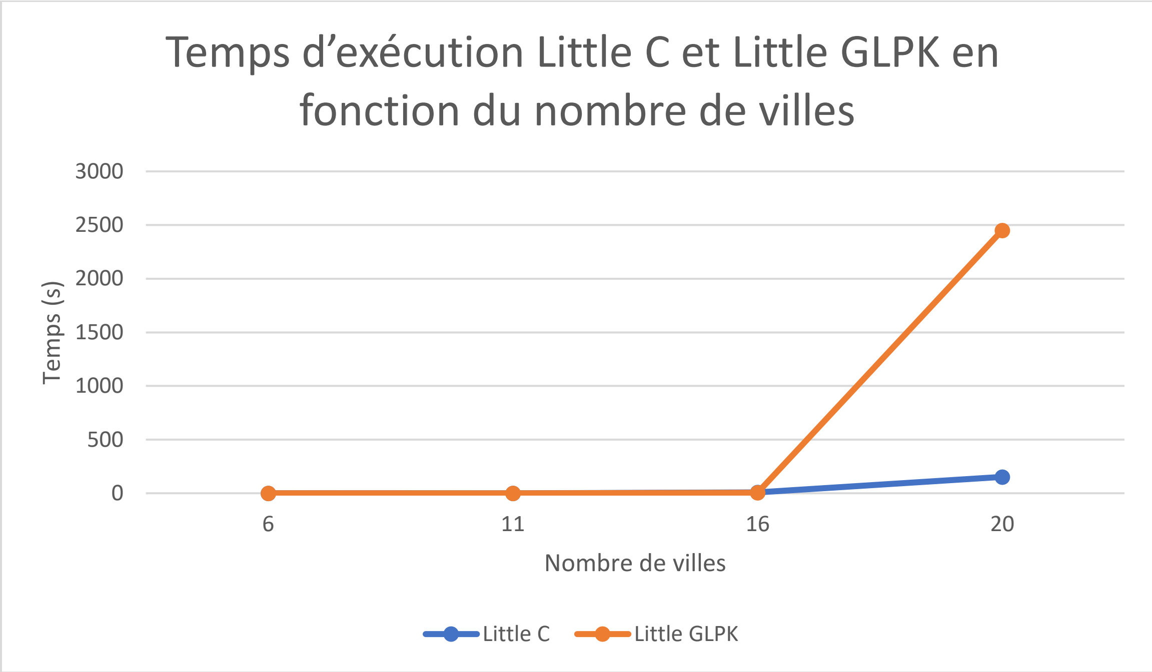
```
INTEGER OPTIMAL SOLUTION FOUND
Time used:   0.1 secs
Memory used: 0.5 Mb (529700 bytes)
Display statement at line 26
(0,1)
(1,6)
(2,7)
(3,5)
(4,0)
(5,4)
(6,2)
(7,8)
(8,9)
(9,3)
Display statement at line 27
2826.49840026796
Model has been successfully processed
```

Concerant les résultats, ils sont sensiblement les même, à la difference pres que la distance optimale est plus précise avec la méthode GLPK que le programme Little en C. Les chemins entre les villes sont indiqués différement mais sont les memes.

Concernant maintenant le temps exécution, en les comparants, on remarque que le programme GLPK est globalement plus rapide jusqu'à 16 villes puis se fait grandement ratrapé par le programme Little C. Pour 21 villes, son temps d'exécution dépasse 1heure (à noté que pour 20 villes nous sommes déjà à 40 minutes), ce qui n'est pas viable pour un programme car son objectif premier est d'etre tres rapide. Le programme Little C est donc plus performant que le GLPK.

<u>Comparaison du temps d'exécution pour le programme Little C et GLPK (jusqu'à 21 villes avec un pas de 5):</u>

	Temps d'exécution (en s)	
Nombre de villes	Little C	Little GLPK
6	0.000278	0.00000001
11	0.009417	0.1
16	8.750216	5.5
20	152.443007	2450
21	153.756221	+1h



4. Optimisation de l’algorithme Little C

Pour optimiser l’algorithme, il « suffit » d’éviter les « sous-tours » qui peuvent se former dans l’algorithme car leur solution ne mène à rien. Pour cela il faut rajouter ces 2 lignes de code dans la fonction Little() :

```
// Little+ Optimisation
if (jzero == 0 && izero == ending_town[iteration - 1]){
    return;
}
```

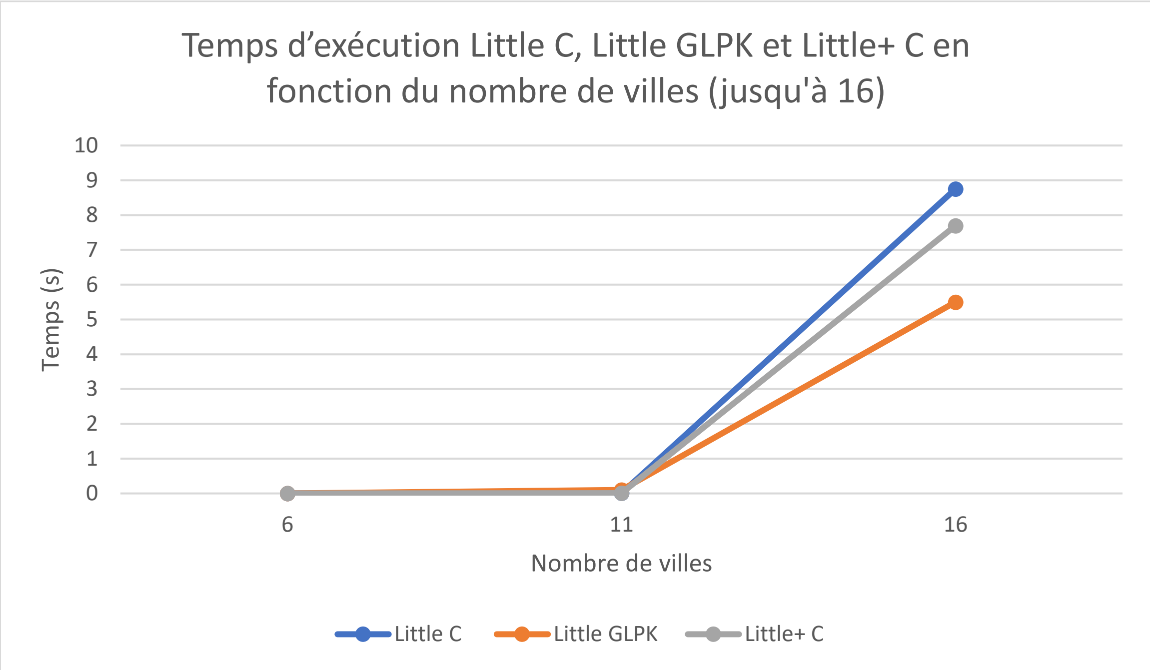
Ici, l’algorithme élimine tôt dans la fonction un grand nombre de branches invalides, ce qui donne un gain de temps dans les calculs.

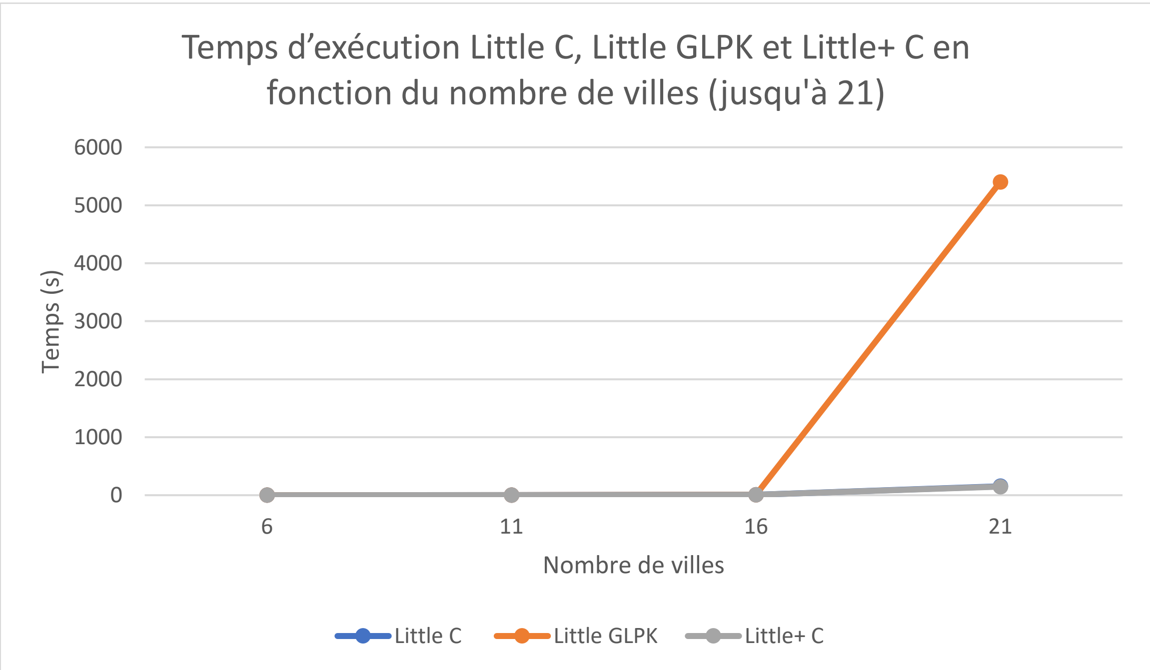
En effet, l'instruction if (jzero == 0 && izero == ending_town[iteration - 1]) return; vérifie si le nœud actuel est un nœud répété, ce qui signifie que la dernière ville d'arrivée est égale à la ville de départ actuelle. Dans ce cas, l'algorithme retourne immédiatement pour éviter d'explorer deux fois le même nœud.

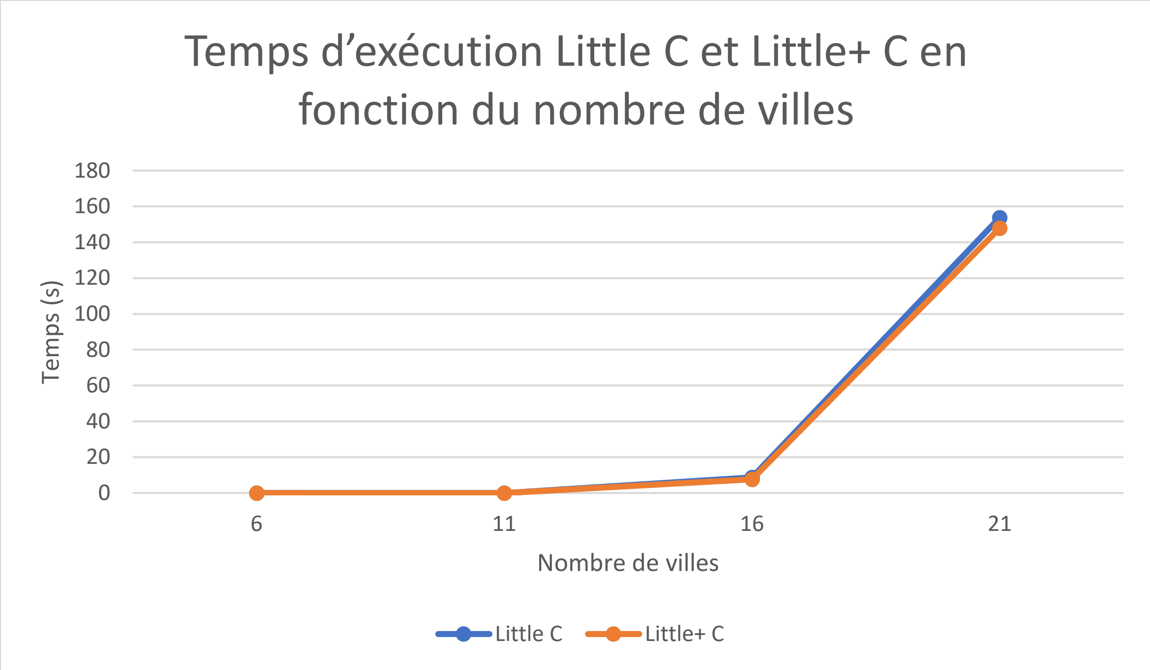
Si nous effectuons ici de nouveau une comparaison entre le programme Little C, Little GLPK et Little+ en C Nous remarquons directement que le Little+ est le plus performant des 3. Cependant, même s’il est plus performant, son temps d’exécution augmente aussi de manière exponentielle et il reste assez proche de Little C classique. Il y a surement d’autre méthodes plus optimisées pour limiter encore plus le temps d’exécution. J’ai eu l’opportunité d’en tester une autre en ajoutant une fonction de recherche de sous tour à la place de ces 2 lignes, mais cette dernière allongeait le temps d’exécution, je n’ai donc pas opté pour cette solution.

<u>Tableau comparatif des temps d'exécution Little C, Little GLPK et Little+ C en fonction du nombre de villes</u>

	Temps d'exécution (en s)			
Nombre de villes	Little C	Little GLPK	Little + C	Solution optimale
6	0.000278	0.00000001	0.000184	2312.15
11	0.009417	0.1	0.009839	4038.44
16	8.750216	5.5	7.694668	4990.46
21	153.756221	+1h	147.864480	5281.53







N.B. Tous les tests ont été réalisé sur une machine virtuelle linux avec 4Go de RAM ce qui a pu impacter (négativement) sur les résultats de l'expérimentation. Le temps d'exécution pourrait être plus rapide sur une machine plus performante.

5. Conclusion

Pour conclure, ce TP m’a permis de réaliser et d’optimiser un algorithme de Little en C pour résoudre le problème du voyageur de commerce (TSP) et de comparer les performances de différentes méthodes d’optimisation du TSP entre le programme classique, le programme optimisé et le programme GLPK.

Cette étude permet de déduire que le programme Little+ C est plus performant car supprime rapidement les branches inutiles à étudier, que Little GLPK se fait rapidement dépasser par les autres programmes (à partir de 16 villes à étudier) même s’il reste toutefois pratique pour de petit problèmes, mais il prend énormément de temps à s’exécuter dans le cas d’un gros pull de données.

A noté que le programme le plus performant prends tout de même plusieurs heures avant de sortir la solution optimale pour les 52 villes, ce qui est énorme, mais qui dans mon cas peut être lié à la machine.