Rapport de projet OS11

Adrien WARTELLE & HABIBALLAH Seddik

27 décembre 2018

Sommaire

Résumé

Ce projet nous a permis d'étudier 3 heuristiques utilisées pour la résolution (approchée) du problème du voyageur du commerce. Ce problème consiste à trouver un chemin de coût minimal (somme des distances/coûts entre les nœuds du chemin) passant par tous les nœuds d'un graphe. Il s'agit d'un problème NP-difficile, c'est-à-dire que pour trouver la solution optimale, il faut (dans le pire cas) énumérer toute les solutions possibles qui sont au nombre de $n!$, n étant le nombre de nœuds. Ainsi pour des problèmes de taille moyenne ou grande, comme ici où l'on prendra n=50, il est nécessaire de faire appel à des heuristiques pour avoir un temps d'exécution qui n'"explose" pas [1]. Nous allons donc étudier 3 heuristiques de construction de tour par insertion de nœuds dans un tour non complet :

1. PPI : Plus Proche Insertion
2. PLI : Plus Lointaine Insertion
3. MI : Meilleure Insertion

1 Description des heuristiques

Les algorithmes par insertion sont des algorithmes qui construisent des tours par ajout itératif de nœuds. Chaque heuristique va sélectionner, à chaque itération, un nœud d'une manière différente :

1. Avec PPI, celui dont la distance au circuit actuel [2] est minimale
2. Avec PLI, celui dont la distance au circuit actuel est maximale
3. Avec MI : celui dont le coût d'insertion au circuit actuel est minimale

Après avoir sélectionner le nœud, chacun de ces algorithmes va insérer celui-ci de manière optimale dans le circuit, c'est-à-dire avec le coût d'insertion minimal. Pour obtenir le coût d'insertion du nouveau après un nœud candidat dans le circuit, il suffit d'additionner la distance entre le nœud candidat et le nouveau nœud et la distance du nouveau nœud et le voisin (dans le circuit) du nœud candidat et ensuite de soustraire la distance entre le nœud candidat et son voisin. On obtient alors la longueur ajouté au circuit si l'on insère le nouveau nœud, ce qui correspond au coût d'insertion.

On peut alors décrire ces algorithmes comme ceci :

1. Autrement dit, que la complexité de l'algorithme de résolution soit polynomial et non exponentiel
2. qui est la distance minimale à l'ensemble des nœuds du circuit actuel

Algorithme 1 : Algorithme PPI : Plus proche insertion

```
   // Déclaration et Initialisation
 1 C_min : Coût minimal d'insertion
 2 D_min ← +∞ : Distance minimale au circuit
 3 ∀i = 1...n, D^i_tour ← +∞, : Distances des nœuds encore à insérer au
     circuit
 4 B : Nœud sélectionné à insérer (B pour "Best")
 5 A ← 0 : Nœud d'insertion de coût minimal (A avant B, ou B voisin de
     A)
   // Recherche du ième point du point à ajouter au chemin
      actuel
 6 pour i allant de 1 à N faire
       // Réinitialiser le coût et la distance minimales
 7     C_min ← +∞
 8     D_min ← +∞
       // Recherche parmi les nœuds qui ne sont pas encore dans
          le circuit
 9     pour j allant de 1 à N faire
10         si j ∉ circuit alors
11             Recalculer la distance D^j_tour du nœud j au circuit en prenant
                en compte le (i-1)-ème nœud ajouté (nœud i − 1 du circuit)
12             Mise à jour de la distance minimale :
                D_min ← min(D(B, node(i − 1)), D(j, node(i − 1)))
13             Mise à jour du meilleur nœud à ajouter :
                B ← arg1min(D(B, node(i − 1)), D(j, node(i − 1)))  [a]
14         fin
15         si j ∈ circuit alors
16             Calculer le coût d'ajout C_j de B après le nœud j dans le
                circuit
17             Mise à jour du coût (minimal) de meilleure insertion :
                C_min ← min(C_A, C_j)
18             Mise à jour du nœud (prédécesseur) de meilleure insertion :
                A ← argmin(C_A, C_j)
19         fin
20     fin
21     Ajouter le nœud B à la suite du nœud A dans le circuit.
22 fin
```

a. node(i-1) est l'index du (i-1)-ème nœud ajouté au circuit et D(,) est la distance entre deux nœuds

Algorithme 2 : Algorithme PLI : Plus lointaine insertion

```
   // Déclaration et Initialisation
 1 C_min : Coût minimal d'insertion
 2 D_max ← 0 : Distance maximale au circuit
 3 ∀i = 1...n, D^i_tour ← +∞, : Distances des nœuds encore à insérer au
     circuit
 4 B : Nœud sélectionné à insérer (B pour "Best")
 5 A ← 0 : Nœud d'insertion de coût minimal (A avant B, ou B voisin de
     A)
   // Recherche du ième point du point à ajouter au chemin
      actuel
 6 pour i allant de 1 à N faire
       // Réinitialiser le coût et la distance minimales
 7     C_min ← +∞
 8     D_max ← 0
       // Recherche parmi les nœuds qui ne sont pas encore dans
          le circuit
 9     pour j allant de 1 à N faire
10         si j ∉ circuit alors
11             Recalculer la distance D^j_tour du nœud j au circuit en prenant
                en compte le (i-1)-ème nœud ajouté (nœud i − 1 du circuit)
12             Mise à jour de la distance maximale :
                D_max ← max(D(B, node(i − 1)), D(j, node(i − 1)))
13             Mise à jour du meilleur nœud à ajouter :
                B ← arg1max(D(B, node(i − 1)), D(j, node(i − 1)))
14         fin
15         si j ∈ circuit alors
16             Calculer le coût d'ajout C_j de B après le nœud j dans le
                circuit
17             Mise à jour du coût (minimal) de meilleure insertion :
                C_min ← min(C_A, C_j)
18             Mise à jour du nœud (prédécesseur) de meilleure insertion :
                A ← argmin(C_A, C_j)
19         fin
20     fin
21     Ajouter le nœud B à la suite du nœud A dans le circuit.
22 fin
```

Algorithme 3 : Algorithme MI : Meilleure Insertion

```
   // Déclaration et Initialisation
1  C_min : Coût minimal d'insertion
2  ∀i = 1...N, j = 1..N, C_{j,i} ← +∞ : Coût d'intégration du nœud j dans le
     circuit après le nœud i
3  B : Nœud sélectionné à insérer (B pour "Best")
4  A ← 0 : Nœud d'insertion de coût minimal (A avant B, ou B voisin de
     A)
   // Recherche du ième point du point à ajouter au chemin
      actuel
5  pour i allant de 1 à N faire
   |  // Réinitialiser le coût minimal
6  |  C_min ← +∞
   |  // Recherche parmi les nœuds qui ne sont pas encore dans
   |     le circuit
7  |  pour j allant de 1 à N faire
   |  |  // Mise à jour des coûts
8  |  |  si j ∉ circuit alors
9  |  |  |  Calculer le coût C_{j,node(i−1)} et recalculer C_{j,prec(node(i−1))} [a]
   |  |  |   qui a changé avec l'ajout du (i-1)-ème nœud
10 |  |  fin
   |  |  // Recherche du coût minimum
11 |  |  Mise à jour du meilleur nœud à ajouter :
   |  |   B ← arg1min_{i=1...N,j=1...N} C_{j,i}
12 |  |  Mise à jour du nœud (prédécesseur) de meilleure insertion :
   |  |   A ← arg2min_{i=1...N,j=1...N} C_{j,i}
13 |  |  Mise à jour du coût (minimal) de meilleure insertion (utilisé
   |  |   pour trouver le minimum) : C_min ← min_{i=1...N,j=1...N} C_{j,i}
14 |  fin
15 |  Ajouter le nœud B à la suite du nœud A dans le circuit.
16 fin
```

a. prec(node(i-1)) donne l'antécédent dans le circuit (différent de l'ordre d'ajout) du (i-1)-ème nœud ajouté

Afin de générer des jeux de tests, nous avons utilisé le générateur pseudo-aléatoire de Mersenne Twister (Makoto Matsumoto et Takuji Nishimura en 1997) initialisé avec 2018 comme racine (SEED). La fonction de génération des points d'entrée a été utilisée pour générer à la suite 100 tests (NB_TEST) de 50 points (NB_POINTS) avec des valeurs allant de 0 à 10 (maxX et maxY) :

Algorithme 4 : Génération pseudo-aléatoire

```
1  pour i allant de 1 à N faire
2  |  Génération pseudo-aléatoire d'un flottant double précision entre 0 et
   |   maxX (resp. maxY) à assigner à la coordonnée x (resp. y) du point
   |   d'entrée :
3  |  input_points(i).x ← generator.randDblExc(maxX)
4  |  input_points(i).y ← generator.randDblExc(maxY)
5  fin
```

2 Résultats et comparaisons

A l'aide de différents fichiers, nous avons pu effectuer des tests sur les 3 algorithmes d'insertion :

- Circuit.h déclare les structures et les variables pour définir et gérer le graphe
- TSPheuristics.cpp contient les algorithmes d'insertion et le programme principal avec les tests
- plots.m (en annexe) a permis d'afficher les exemples graphiques
- stats.R a permis de calculer les statistiques avec les données de sortie du programme (dans "./output/statistics.csv")

Afin de vérifier le bon fonctionnement des algorithmes, nous avons effectué un petit test sur un graphe (décrit dans "./input/input_test.txt") et nous avons tracé les solutions à l'aide de Matlab sur les figures 1, 2 et 3.

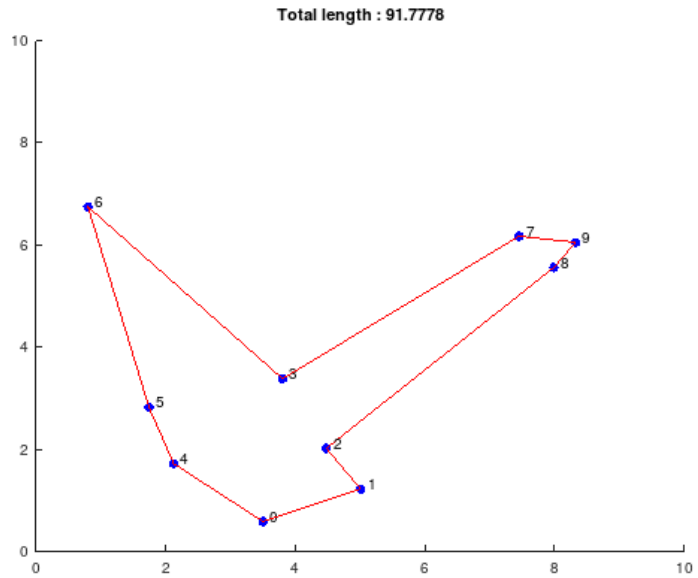

FIGURE 1 – Chemin de l'algorithme PPI

Les numéros des points correspondent à leur ordre d'ajout dans le chemin. On voit ainsi que PPI prend bien à chaque fois le nœud le plus proche du circuit et PLI le plus loin. On peut noter que MI donne le même résultat que PPI ici mais on voit bien que l'ordre d'ajout est différent. Il n'est pas rare que le point de coût d'insertion minimal et le point le plus proche du graphe coïncident : il suffit pour cela qu'on ait un point hors circuit qui soit de distance minimal avec un point du graphe et qu'il soit suffisamment proche avec un autre point pour avoir un coût d'insertion minimal.

On constate que, sur cet exemple, PPI et MI sont plus performants avec une longueur d'environ 92 que PLI avec une longueur de 95 mais ce n'est qu'anecdotique. En effet pour tester les algorithmes, on doit utiliser plusieurs tests, ici 100, et plus de points, ici 50. On obtient alors les statistiques du tableau 1.

On voit ainsi que les algorithmes donne des performance similaires puisque

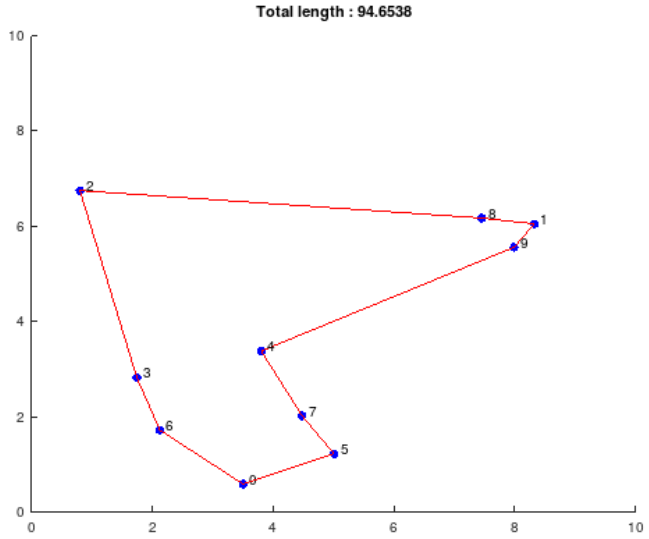

FIGURE 2 – Chemin de l'algorithme PLI

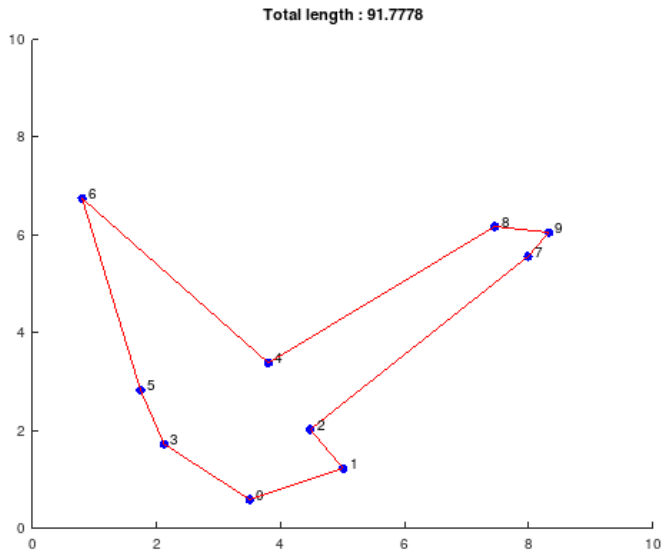

FIGURE 3 – Chemin de l'algorithme MI

les longueurs moyennes obtenues sont entre 119.75 et 120.8. On peut remarquer que PLI possède la meilleure performance moyenne mais que c'est PPI qui est le plus souvent premier (38 fois sur 100). Grâce à l'utilisation d'un tableau de coût et d'une mise à jour intelligente ce celui-ci, le temps d'exécution est minimale avec MI (0.23 ms) tandis qu'il est maximal avec PLI (0.36 ms) notamment car on gère 2 tableaux et on n'effectue pas une mise à jour dynamique d'un tableau de coût d'insertion.

Algorithme	PPI	PLI	MI
Temps d'exécution moyen (ms)	0.3582573	0.2699245	0.2307893
Longueur moyenne	120.7714	119.7717	120.2111
Nombre de fois 1er	38	31	31
Nombre de fois 2nd	31	34	35
Nombre de fois 3ème	31	35	34

TABLE 1 – Table des statistiques

Les histogrammes des longueurs obtenus sur les figures 1, 2 et 3 montrent que les distributions sont assez proches de distributions normales avec des moyennes et des écarts-types qui sont assez similaires (entre 11.8 et 12.4). Il y a juste la distribution de PLI qui semble avoir une concentration un peu plus forte autour de 120.

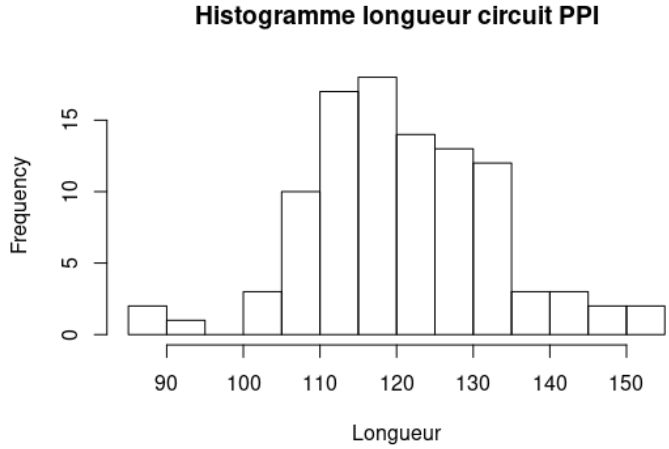

FIGURE 4 – Histogramme PPI

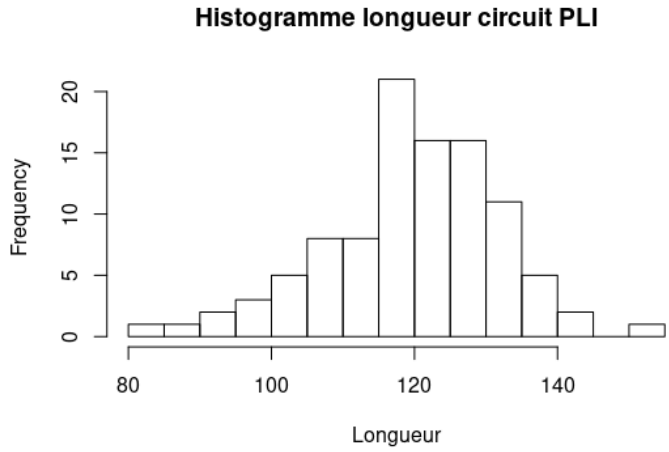

FIGURE 5 – Histogramme PLI

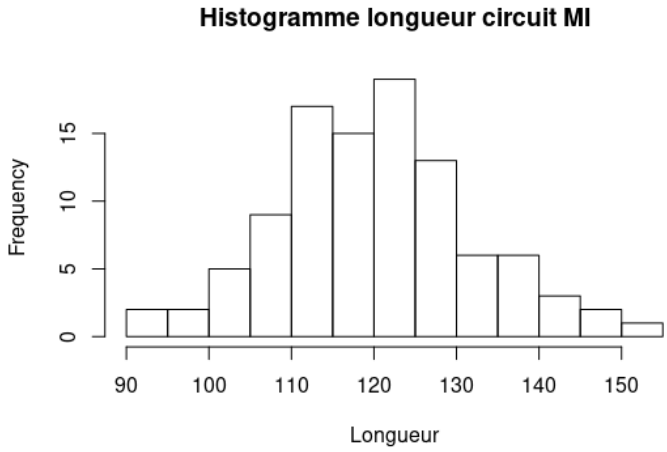

FIGURE 6 – Histogramme MI

3 Annexes

3.1 Visualisation de tournées (plots.m)

```
clear;

fin = './input/input_test.txt';
fout = './output/output_PPI_test.txt';
%'./output/output_PPI_test.txt'
%'./output/output_PLI_test.txt'
%'./output/output_MI_test.txt'

% read input
f=fopen(fin);
tline = fgetl(f);
tlines = cell(0,1);
while ischar(tline)
    tlines{end+1,1} = tline;
    tline = fgetl(f);
end
fclose(f);

n = sscanf(tlines{1}, '%d'); %number of points
max = sscanf(tlines{2}, '%f %f'); %maximum X and Y coordinates
p = zeros(n, 2); %points
for i=1:n
    p(i,:) = sscanf(tlines{i+2}, '%f %f');
end;

% read output
f=fopen(fout);
oline = fgetl(f);
olines = cell(0,1);
while ischar(oline)
    olines{end+1,1} = oline;
    oline = fgetl(f);
end
fclose(f);

starts = zeros(n, 2); %circuit connections
number = zeros(n, 1); %number of when the node was added
for i=1:n
    l = sscanf(olines{i}, '%f %f %d');
    starts(i,1)=l(1);
    starts(i,2)=l(2);
    number(i)=l(3);
end
total_length = sscanf(olines{n+1}, '%f');

% plot
figure; ##close all;
hold on;
axis([0 max(1) 0 max(2)]);
scatter(p(:,1),p(:,2),30,'b','filled');
title(["Total length : " , num2str(total_length)]);
for i=1:n-1
    text(starts(i,1)+0.1,starts(i,2)+0.1, num2str(number(i)));
    plot([starts(i,1) starts(i+1,1)],[starts(i,2) starts(i+1,2)],'r');
end
text(starts(n,1)+0.1,starts(n,2)+0.1, num2str(number(n)));
plot([starts(n,1) starts(1,1)],[starts(n,2) starts(1,2)],'r');

hold off;
```

3.2 Visualisation de statistiques (stats.R)

```
meanExecPPI = mean(statistics$ExecPPI)
meanExecPLI = mean(statistics$ExecPLI)
meanExecMI = mean(statistics$ExecMI)
```

```
meanLengthPPI = mean(statistics$LengthPPI)
meanLengthPLI = mean(statistics$LengthPLI)
meanLengthMI = mean(statistics$LengthMI)
hist(statistics$LengthPPI,nclass=12,
     main= 'Histogramme longueur circuit PPI',
     xlab= 'Longueur',
     ylab= 'Frequency')
hist(statistics$LengthPLI,nclass=12,
     main= 'Histogramme longueur circuit PLI',
     xlab= 'Longueur',
     ylab= 'Frequency')
hist(statistics$LengthMI,nclass=12,
     main= 'Histogramme longueur circuit MI',
     xlab= 'Longueur',
     ylab= 'Frequency'))
scores <- c(c(0,0,0),c(0,0,0),c(0,0,0))
dim(scores) <- c(3,3)
for(n in 1:100) {
  if(statistics$LengthPPI[n] > statistics$LengthPLI[n]) {
    if(statistics$LengthPPI[n] > statistics$LengthMI[n]) {
      scores[1,1] = scores[1,1]+1;
      if(statistics$LengthPLI[n] > statistics$LengthMI[n]) {
        scores[2,2] = scores[2,2]+1;
        scores[3,3] = scores[3,3]+1;
      } else {
        scores[2,3] = scores[2,3]+1;
        scores[3,2] = scores[3,2]+1;
      }
    } else {
      scores[1,2] = scores[1,2]+1;
      scores[2,3] = scores[2,3]+1;
      scores[3,1] = scores[3,1]+1;
    }
  } else {
    if(statistics$LengthPLI[n] > statistics$LengthMI[n]) {
      scores[2,1] = scores[2,1]+1;
      if(statistics$LengthPPI[n] > statistics$LengthMI[n]) {
        scores[1,2] = scores[1,2]+1;
        scores[3,3] = scores[3,3]+1;
      } else {
        scores[1,3] = scores[1,3]+1;
        scores[3,2] = scores[3,2]+1;
      }
    } else {
      scores[1,3] = scores[1,3]+1;
      scores[2,2] = scores[2,2]+1;
      scores[3,1] = scores[3,1]+1;
    }
  }
}
```