PROJET: PROBLÈME DU VOYAGEUR DE COMMERCE - TSP

RAPPORT

Ayoub Abraich
M2 Data Science
Université Evry Val-d'Essonne - Paris Saclay
ayoub.abraich@etud.univ-evry.fr

September 26, 2019

1 Introduction

Je me base principalement dans ce projet sur la thèse [1] et l'article [2].

1.1 Etat de l'art

Le problème du voyageur de commerce (TSP : Travelling Salesperson Problem) est l'un des problèmes d'optimisation combinatoire les plus répandus. Étant donné un graphe complet $G = (V, E)$ et une fonction de poids $w : E \to \mathbb{N}$.Le but est de trouver un cycle Hamiltonien dans G (également appelé un tour) de poids minimum. C'est l'un des problèmes centraux de l'informatique et de la recherche opérationnelle. Il est bien connu d'être NP-difficile et a fait l'objet de recherches selon différentes perspectives, notamment par approximation, algorithmes de temps exponentiel et heuristiques.

En pratique, le TSP est souvent résolu au moyen d'heuristiques de recherche locales, dans lesquelles on part d'un cycle Hamiltonien arbitraire en G, puis on modifie le cycle au moyen de modifications locales en une série d'étapes. Après chaque étape, le poids du cycle devrait s'améliorer; lorsque l'algorithme ne trouve aucune amélioration, il s'arrête. L'un des exemples les plus réussis de cette approche est l'heuristique $k-$opt, dans laquelle un $k-$mouvement amélioré est effectué à chaque étape. Soit un cycle Hamiltonien H dans un graphe $G = (V, E)$ un $k-$mouvement est une opération qui supprime k arêtes de H et ajoute k arêtes de G de sorte que l'ensemble des arêtes résultant H' est un nouveau cycle Hamiltonien. Le $k-$mouvement s'améliore si le poids de H' est plus petit que le poids de H.

Les lignes d'attaque traditionnelles pour les problèmes NP-difficiles sont les suivantes:

- Concevoir des algorithmes exacts, qui fonctionnent raisonnablement vite que pour des problèmes de petite taille.
- Concevoir des algorithmes "sous-optimaux" ou heuristiques, c'est-à-dire des algorithmes qui fournissent des solutions approchées dans un délai raisonnable.
- Recherche de cas spéciaux pour le problème ("sous-problèmes") pour lesquels des heuristiques optimales ou exactes sont possibles.

1.2 Algorithmes exacts

La solution la plus directe serait d'essayer toutes les permutations (combinaisons ordonnées) et de voir laquelle est la moins chère (en utilisant la recherche par force brute). Le temps d'exécution a un compléxité de l'ordre $O(n!)$, la factorielle du nombre de villes, de sorte que cette solution devient impraticable, même pour seulement 20 villes.

L'une des premières applications de la programmation dynamique est l'algorithme de Held – Karp qui résout le problème dans le temps $O\left(n^2 2^n\right)$. Cette compléxité a également été atteinte par Exclusion-Inclusion dans une tentative précédant l'approche de programmation dynamique. Améliorer ces délais semble difficile. Par exemple, il n'a pas été déterminé si un algorithme exact pour TSP qui s'exécute dans le temps $O\left(1.9999^n\right)$ existe [Wikipédia].

Les autres approches comprennent:

- Divers algorithmes de branch-and-bound, qui peuvent être utilisés pour traiter les TSP contenant 40 à 60 villes.
- Algorithmes d'amélioration progressive utilisant des techniques rappelant la programmation linéaire. Fonctionne bien jusqu'à 200 villes.
- Implémentations de branch-and-bound et problem-specific cut generation (branch-and-cut) : c'est la méthode de choix pour résoudre les instances volumineuses. Cette approche tient le record actuel, résolvant une instance avec 85 900 villes, voir Applegate et al. (2006).

Une solution exacte pour 15 112 villes allemandes de TSPLIB a été trouvée en 2001 en utilisant la méthode du plan de coupe (cutting-plane method) proposée par George Dantzig, Ray Fulkerson et Selmer M. Johnson en 1954, basée sur la programmation linéaire. Les calculs ont été effectués sur un réseau de 110 processeurs situés à l'Université Rice et à l'Université Princeton.

Le temps de calcul total était équivalent à 22,6 ans sur un seul processeur Alpha 500 MHz. En mai 2004, le problème TSP , qui consistait à se rendre dans les 24 978 villes de Suède, était résolu: un circuit long d'environ 72 500 kilomètres avait été découvert et il était prouvé qu'il n'existait pas de circuit plus court . En mars 2005, le problème rencontré par le vendeur voyageur lors de la visite des 33 810 points d'un circuit imprimé a été résolu avec Concorde TSP Solver: un circuit de 66 048 945 unités a été trouvé et il a été prouvé qu'il n'existait pas de circuit plus court. Le calcul a pris environ 15,7 années CPU (Cook et al. 2006). En avril 2006, une instance avec 85 900 points a été résolue à l'aide de Concorde TSP Solver, ce qui représente plus de 136 années d'UC, voir Applegate et autres (2006).

1.3 Algorithmes heuristiques et d'approximation

Divers heuristiques et algorithmes d'approximation, qui ont rapidement permis de trouver de bonnes solutions, ont été conçus. Les méthodes modernes peuvent trouver des solutions à des problèmes extrêmement importants (des millions de villes) dans un délai raisonnable, avec une probabilité de 2 à 3% de la solution optimale .

Plusieurs catégories d'heuristiques sont reconnues comme : nearest neighbor (NN), algorithme de Christofides,Pairwise exchange,heuristique K-opt,Heuristique V-opt,Algorithmes de chaîne de Markov optimisés,Ant colony optimization etc..

1.4 Plan du projet

Nous allons présenter quelques notions liés au problème TSP , puis nous étudions rapidement les algorithmes : Naif (Brute force) & Programmation dynamique. Ensuite, nous allons les implémenter en R & Rcpp et enfin nous allons comaprer les résultats obtenues avec le Benchmark.

2 Formalisation

2.1 Notions

Définition 2.1 ($\mathcal{NP}$) *La classe $\mathcal{NP}$ est l'ensemble des problèmes de décision $L \subseteq \Sigma^*$ pour lesquels il existe un algorithme polynomial $\mathcal{V}$ tel que pour tout $x \in \Sigma^*$, on a :*
$x \in L$ si et seulement s'il existe $y \in \Sigma^*$ de taille polynomiale en $|x|$ tel que $\mathcal{V}$ retourne « oui » sur l'entrée (x, y).
La chaîne y est alors appelé certicat polynomial. Pour TSP , un certicat sera une permutation de $\{1, \cdots, n\}$.

Définition 2.2 (TSP) *Le $\mathrm{TSP} = \{(G, w, t)\}$ peut être décrit comme suit:*

- *$G = (V, E)$ où G est un graphe complet*
- *Une fonction de poids $w : E \to \mathbb{N}$.*
- *$t \in Z$*
- *G est un graphe contenant un circuit (tour) avec un coût ne dépassant pas t.*

Définition 2.3 (Cycle Hamiltonien) *Un cycle Hamiltonien est un cycle dans un graphe passant par tous les sommets une fois.*

2.2 Problème formel

On peut décrire formellement le problème TSP comme suit :

TRAVELING SALESMAN PROBLEM
Entrée : Un ensemble de villes $V = \{v_1, \cdots, v_n\}$, une distance $d(i,j)$ pour toute paire $i, j \in \{1, \cdots, n\}$, $i \neq j$
Sortie : Une permutation ϕ de $\{1, \cdots, n\}$
But : Minimiser $\left(\sum_{i=1}^{n-1} d(\phi(i), \phi(i+1))\right) + d(\phi(n), \phi(1))$

2.3 Complexitées

- Complexité de TSP naif : $\mathcal{O}(n!)$
- Complexité de TSP DP : $\mathcal{O}(n^2 \cdot 2^n)$
- Comparaison de temps d'exécution selon la complexité de l'algorithme, pour une machine éffectuant un million d'opérations par seconde :

	Taille de l'instance (n)				
Fonction	20	30	40	50	60
n	0,00002 sec.	0,00003 sec.	0,00004 sec.	0,00005 sec.	0,00006 sec.
n^2	0,0004 sec.	0,0009 sec.	0,0016 sec.	0,0025 sec.	0,0036 sec.
n^3	0,008 sec.	0,27 sec.	0,064 sec.	0,125 sec.	0,216 sec.
n^5	3,2 sec.	24,3 sec.	1,7 min.	5,2 min.	13 min.
2^n	1 sec.	17,9 min.	12,7 jours	35,7 jours	366 siècles
3^n	58 min.	6,5 années	3855 siècles	2×10^8 siècles	$1{,}3 \times 10^{13}$ siècles

3 Algorithmes

3.1 Algorithme TSP Naif

Algorithm 1 TSP Naif

Require: Nombre de villes n et une liste de coûts $c(i,j)_{i,j=1\cdots n}$ (Nous commençons par la ville numéro 1).
Ensure: Vecteur de villes et coût total

(* Valeurs de départ *)
$C = 0$
$Cost = 0$
$Visites = 0$
$e = 1$ (* e = pointeur de la ville visitée)
(* détermination du tour et du coût)
for $r = 1$ to $n - 1$ **do**
    choisir le pointeur j avec :
    minimum $= c(e,j) = \min\{c(e,k); visites(k) = 0\, et\, k = 1,\ldots,n\}$
    $cost = cost + minimum$
    $e = j$
    $C(r) = j$
end for
$C(n) = 1$
$cost = cost + c(e,1)$

Figure 1: Pseudocode de TSP Naif

3.2 Algorithme TSP DP : Programmation dynamique ou Bellman-Held–Karp

La programmation dynamique (généralement appelée DP) est une technique très puissante pour résoudre une classe de problèmes particulière. Cela demande une formulation très élégante de l'approche et une pensée simple et la partie codage est facile. L'idée est simple : si vous avez résolu un problème avec l'entrée donnée, conservez le résultat pour référence ultérieure afin d'éviter de résoudre à nouveau le même problème. Si le problème donné peut être divisé en sous-problèmes plus petits et que ces sous-problèmes plus petits sont à leur tour divisés en problèmes encore plus petits, et dans ce processus. En outre, les solutions optimales aux sous-problèmes contribuent à la solution optimale du problème donné. Il existe deux façons de le faire :

- De haut en bas (Top-Down) : Commencez à résoudre le problème en le décomposant. Si vous voyez que le problème a déjà été résolu, renvoyez la réponse enregistrée. Si cela n'a pas été résolu, résolvez-le et enregistrez la réponse. Ceci est généralement facile à penser et très intuitif. Ceci est appelé Memoization.
- De bas en haut (Bottom-Up): Analysez le problème et voyez l'ordre dans lequel les sous-problèmes sont résolus et commencez à résoudre du sous-problème trivial jusqu'au problème donné. Dans ce processus, il est garanti que les sous-problèmes sont résolus avant de résoudre le problème. Ceci est appelé programmation dynamique.

L'algorithme de Bellman-Held-Karp repose sur une optimisation niveau par niveau. En effet, le cœur de la méthode peut être décrit par une simple équation récursive. Pour le configurer, notons x un point de départ fixe pour les visites et pour toute paire des villes (i,j) notons $dist(i,j)$ le coût du trajet de i à j. Pour tout S tel que $x \notin S$ et pour tout $t \in S$, notons $opt(S,t)$ le coût minimum d'un chemin commençant par x, passant par tous les points de S et se terminant par t. On a :

$$opt(S,t) = \min(opt(S\backslash\{t\},q) + dist(q,t) : q \in S\backslash\{t\}) \tag{1}$$

De plus, si N est l'ensemble de toutes les villes autres que x, alors la valeur optimale du TSP est :

$$\nu^* = \min(opt(N,t) + dist(t,x) : t \in N) \tag{2}$$

Notez que pour tout $q \in N$ nous avons $opt(\{q\},q) = dist(x,q)$. À partir de ces valeurs, l'équation récursive [] est utilisée pour construire les valeurs $opt(S,t)$ pour tous les $S \subseteq N$ et $t \in S$, en parcourant des ensembles à deux éléments, puis des ensembles à trois éléments, puis par étape jusqu'à l'ensemble complet N. Une fois que nous avons

les valeurs $opt(N,t)$ pour tout $t \in N$, nous utilisons [] pour trouver ν^*. Maintenant, dans un second passage, le tour optimal est calculé en identifiant d'abord une ville v_{n-1} telle que $opt\,(N, v_{n-1}) + dist\,(v_{n-1}, x) = \nu^*$, puis identifier une ville $v_{n-2} \in N \setminus \{v_{n-1}\}$ telle que $opt\,(N \setminus \{v_{n-1}\}, v_{n-2}) + dist\,(v_{n-1}, v_{n-2}) = opt\,(N, v_{n-1})$ et ainsi de suite jusqu'à ce que nous ayons v_1. Le tour optimal $(x, v_1, \ldots, v_{n-1})$. Cette seconde passe doit permettre à l'algorithme de ne stocker que les valeurs $opt(S,t)$ et non les chemins réels qui déterminent ces valeurs.

La durée d'exécution étant proportionnelle à $n^2 2^n$, nous ne résoudrons donc pas les instances de test dans plus de 30 villes. Comme nous le verrons cependant, le code est compétitif sur les petites instances par rapport à la version naive de TSP.

Algorithm 2 : Dynamic Programming algorithm for the original TSP

Data: A set of locations V, an arbitrary location $v \in V$ and cost function c

Result: A shortest tour that visits all locations in V

```
1  Initialize D_TSP with values ∞ ;
2  Initialize a table P to retain predecessor locations ;
3  Initialize v as an arbitrary location in V ;
4  foreach w ∈ V do
5      D_TSP({w}, w) ← c(v, w) ;
6      P({w}, w) ← v ;
7  for i = 2, ..., |V| do
8      for S ⊆ V where |S| = i do
9          foreach w ∈ S do
10             foreach u ∈ S do
11                 z ← D_TSP(S \ {w}, u) + c(u, w) ;
12                 if z < D_TSP(S, w) then
13                     D_TSP(S, w) ← z ;
14                     P(S, w) ← u ;
15 return path obtained by backtracking over locations in P starting at P(V, v) ;
```

Figure 2: Pseudocode de Bellman-Held-Karp

4 Implémentations

4.1 Algorithme TSP Naif

4.1.1 Implémentation avec R

1. Simulation des n villes :

```
villes <- function(n){
      max_x <- 20
      max_y <- 20
      set.seed(9999779)
      villes <- data.frame(id = 1:n, x = runif(n, max = max_x),
             y = runif(n, max = max_y))
      return(villes)
}
plot_villes<-function(n){ggplot(villes(n), aes(x, y)) + geom_point()}
```

2. Fonction 'distance' : calcule la matrice (symétrique) des distances entre les n villes :

```
distance <- function(n) {
  return(as.matrix(stats::dist(select(villes(n), x, y), diag = TRUE, upper = TRUE)))
}
```

3. Fonction 'optimize':
Input : cost_list - la liste des couts totals des (n-1)! permutations ; path_list - la liste des paths correspondents aux couts de 'cost_list'
Output : c - Cout total minimal path - Path minimal correspondent à c

```
optimize<-function(cost_list,path_list){
  c=as.numeric(min( unlist(cost_list)))
  order=match(c,cost_list)
  path=path_list[order]
  return(c(c,path))
}
```

4. Fonction 'cout_total' : Input : p permutation (chemin) , par exemple : p=(2,4,3,5) pour n=6
Output : Cout total du chemin p

```
cout_total<-function(p,dist_fun){

c<-dist_fun(1,p[1])

i=1
l=list(dist_fun(1,p[1]))
while (i<=length(p))
{
    a=p[i]
    b=p[i+1]
    l<-c(l,list(dist_fun(a,b)))
    i<-i+1
}
l<-c(l,list(dist_fun(p[length(p)],1)))
l=unlist(l[!is.na(l)])

  return(sum(l))
}
```

5. Foncion 'TSP_Naif_R' : Input : N villes
Output : Cout minimal et chemin correspondent

```
TSP_Naif_R<- function(N){
  dist_fun <- function(i, j) {vapply(seq_along(i), function(k) distance(N)
        [i[k], j[k]], numeric(1L))}
  perm_list=permutations(n = N-1,r=N-1, v = 2:N)
```

```
  cost_list=list()
  path_list=list()
for (i in seq(1,dim(perm_list)[1])) {
    #progress(i)
    p=perm_list[i,]
    cost_list=c(cost_list,cout_total(p,dist_fun))
    path_list=c(path_list, list(c(1,p,1)))
}
  res=optimize(cost_list,path_list)
  print(c('Le cout minimal est:',res[1]))
  print(c('Le chemin le plus court est :',res[2]))
  return(res)
}
T0 <- Sys.time()
TSP_Naif_R(7)
T1 <- Sys.time()
print(c("Time (s)=",T1 - T0))
>>>
[[1]]
[1] "Le cout minimal est:"

[[2]]
[1] 47.00368

[[1]]
[1] "Le chemin le plus court est :"

[[2]]
[1] 1 3 7 4 6 2 5 1
[1] "Time (s)="            "24.0072641372681"
```

4.1.2 Implémentation avec C++

1. Fichier 'TSP_NAIF_CPP.cpp' :

```
#include <bits/stdc++.h>
#include <iostream>
#include <vector>

#include "RcppArmadillo.h"
using namespace std;
using namespace arma;  // Evite d'écrire arma::fonctionArmadillo

// [[Rcpp::depends(RcppArmadillo)]]

// [[Rcpp::export]]

float TSP_Naif_CPP(int N,mat distance)
{
  // stocker tous les chemins en dehors des chemins sources
  vector<int> paths;
  for (int i = 0; i < N; i++)
    int path_order=0
    if (i != 0)
      paths.push_back(i);

    // store minimum weight Hamiltonian Cycle.
    float min_path = 1000.;
    do {

```

```
26        // stocker le poids actuel du trajet (coût)
27        float current_pathweight = 0.;
28
29        // calculer le poids du trajet actuel
30        int k = 0;
31        for (int i = 0; i < (int)paths.size(); i++) {
32          current_pathweight += distance(k,paths[i]);
33          k = paths[i];
34        }
35        current_pathweight += distance(k,0);
36
37        // update minimum
38        min_path = min(min_path, current_pathweight);
39        path_order+=1
40
41      } while (next_permutation(paths.begin(), paths.end()));
42
43      return min_path,path_order;
44  }
```

2. Chargement du fichier source Rcpp :

```
1  sourceCpp("TSP_NAIF_CPP.cpp")
2  ## Appel de la fonction TSP_Naif_CPP
3  T0 <- Sys.time()
4  n=7
5  print(c("Le cout minimal =",TSP_Naif_CPP(n,distance(n))))
6  T1 <- Sys.time()
7  print(c("Time (s) =",T1 - T0))
8  >>>
9  [1] "Le cout minimal =" "47.0036773681641"
10 [1] "Time (s) ="        "0.248862981796265"
```

4.2 Implémentation de TSP DP avec C++

1. Fichier 'TSP_DP_CPP.cpp':

```
1  #include <bits/stdc++.h>
2  #include <iostream>
3  #include <vector>
4
5  #include "RcppArmadillo.h"
6  using namespace std;
7  using namespace arma;  // Evite d'écrire arma::fonctionArmadillo
8  // [[Rcpp::depends(RcppArmadillo)]]
9
10 // [[Rcpp::export]]
11 float tsp(int N,mat distance_mat, int pos, int visited, vector<vector<float>>& state)
12 {
13   if(visited == ((1 << N) - 1))
14     return distance_mat(pos,0); // return to starting city
15
16   if(state[pos][visited] != 1000)
17     return state[pos][visited];
18
19   for(int i = 0; i < (int)N; ++i)
20   {
21     // can't visit ourselves unless we're ending & skip if already visited
22     if(i == pos || (visited & (1 << i)))
23       continue;
24
```

```
    float distance = distance_mat(pos,i) + tsp(N,distance_mat, i, visited | (1 << i), state);
    if(distance < state[pos][visited])
      state[pos][visited] = distance;
  }

  return state[pos][visited];
}

// [[Rcpp::export]]

float TSP_DP_CPP(int N,mat distance_mat)
{

  vector<vector<float>> state(N);
  for(auto& neighbors : state)
    neighbors = vector<float>((1 << N) - 1, 1000);

  float cout_minimal=tsp(N,distance_mat, 0, 1, state);

    return cout_minimal;
}
```

2. Chargement du fichier source Rcpp :

```
sourceCpp("TSP_DP_CPP.cpp")
## Appel de la fonction TSP_DP_CPP :
T0 <- Sys.time()
n=7
print(c("Le cout minimal =",TSP_DP_CPP(n,distance(n))))
T1 <- Sys.time()
print(c("Time (s) =",T1 - T0))
>>>
[1] "Le cout minimal =" "47.0036773681641"
[1] "Time (s) ="         "0.0289809703826904"
```

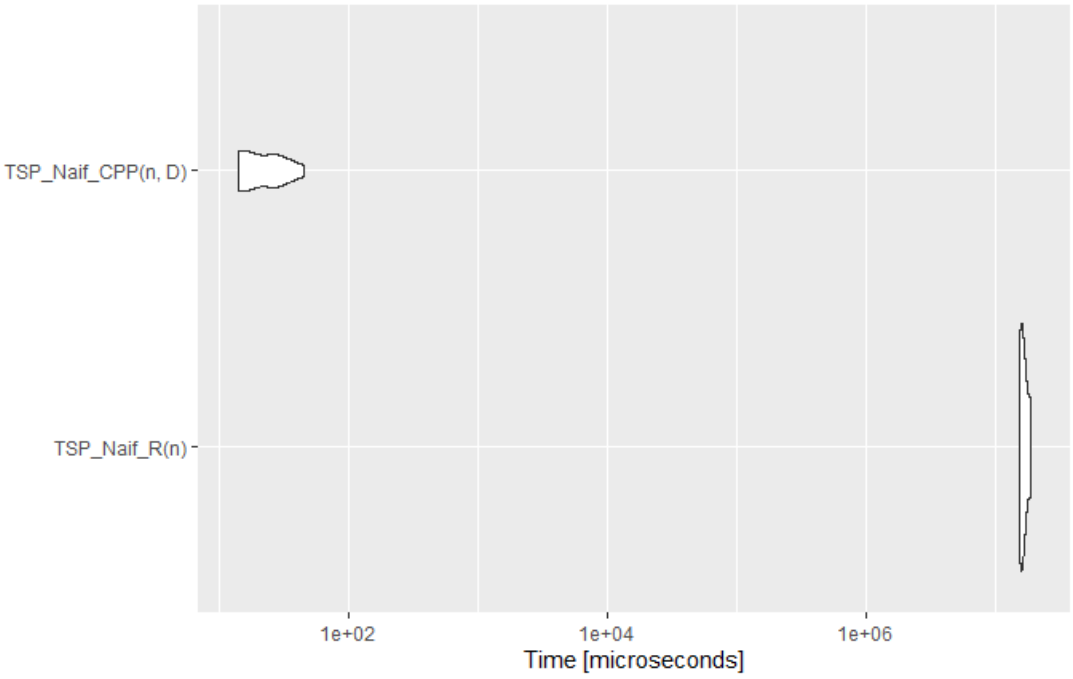
5 Benchmark

5.1 TSP Naif avec R & TSP Naif avec C++

1. Fonction 'benchmark_naif' : affiche le banchmark entre les deux versions naives R & C++ :

```
benchmark_naif <- function(n){
  D=distance(n)
  tm <- microbenchmark(TSP_Naif_R(n),TSP_Naif_CPP(n,D), times =100L)
  autoplot(tm)
}
```

2. Pour $n = 7$:



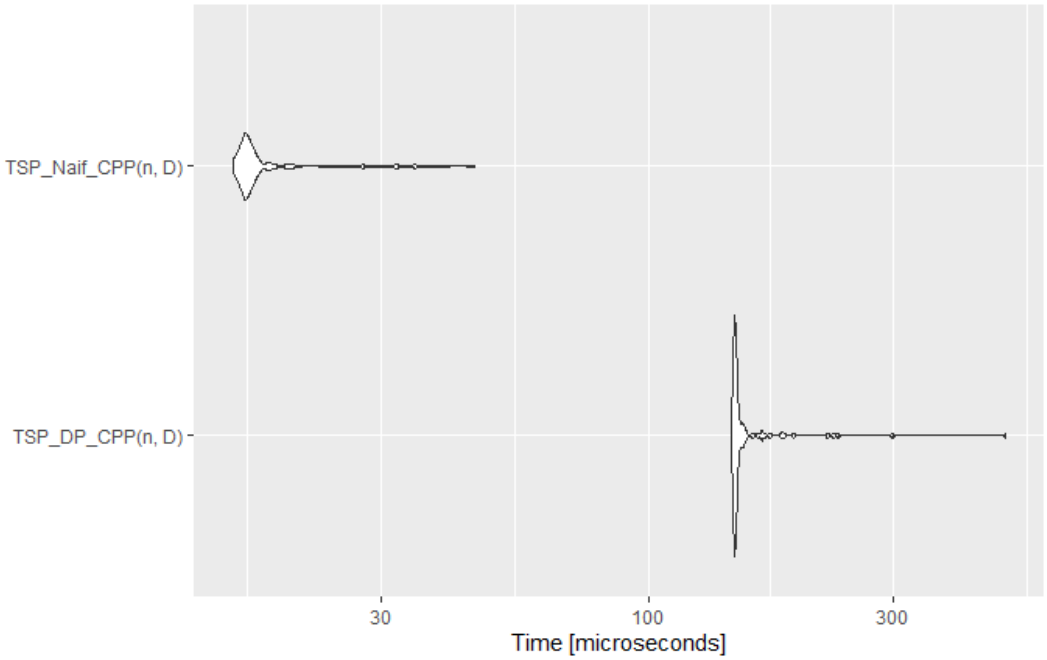
3. Remarque : à partir de $n = 8$, le temps de calcul de TSP_Naif_R explose , pourtant le calcul de TSP_Naif_CPP reste raisonable jusqu'à $n = 15$.

5.2 TSP Naif avec C++ & TSP DP avec C++

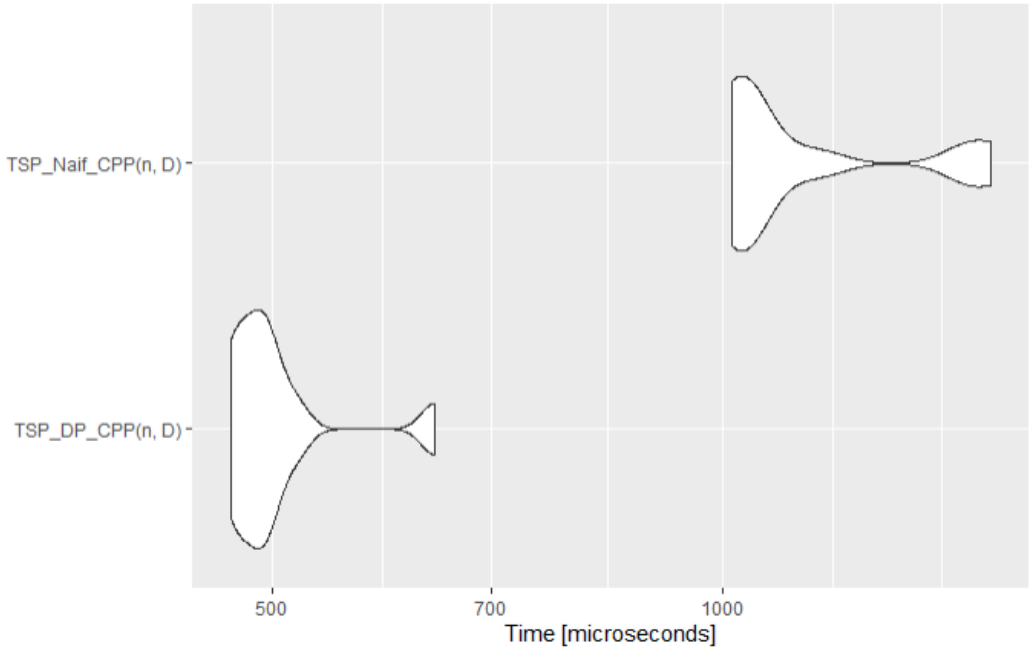
1. Fonction 'benchmark_cpp' : affiche le banchmark entre TSP Naif avec C++ & TSP DP avec C++:

```
benchmark_cpp <- function(n){
  D=distance(n)
  tm <- microbenchmark(TSP_DP_CPP(n,D),TSP_Naif_CPP(n,D), times =100L)
  autoplot(tm)
}
```

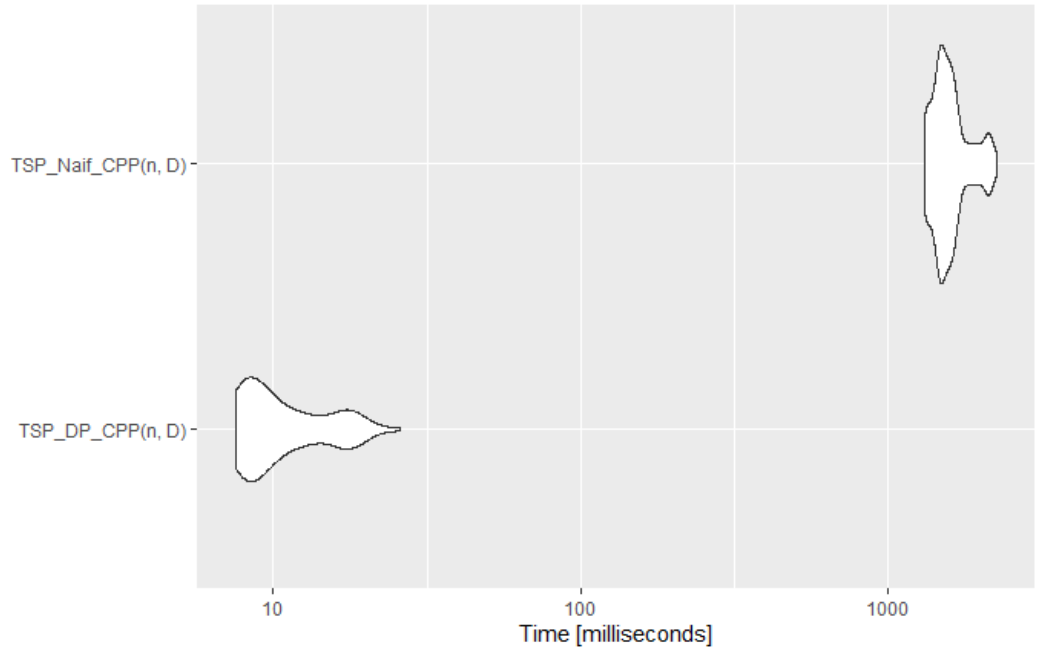
2. Pour $n = 7$:



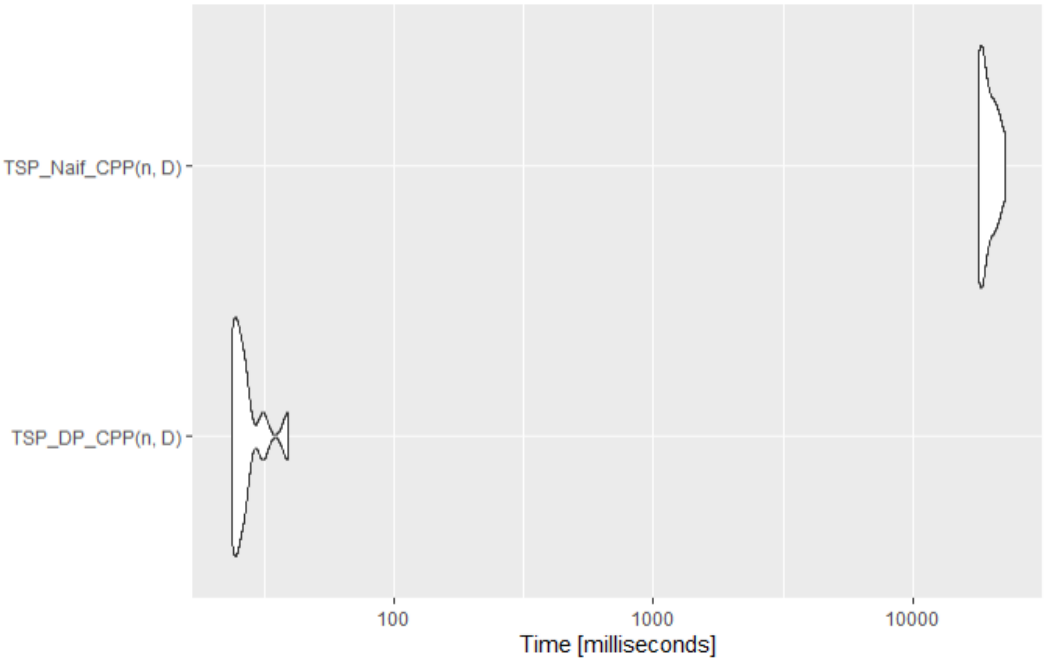
3. Pour $n = 9$:



4. Pour $n = 12$:



5. Pour $n = 13$:



6. Performances : TSP_Naif_CPP

```
temps_cpp<-function(f,n){
  s0 <- Sys.time()
  f(n,distance(n))
  s1 <- Sys.time()
  print(c("Nombre de Villes:",n,"Time (s)=",s1 - s0))
  }

for (n in 5:14){temps_cpp(TSP_Naif_CPP,n)}

>>>
[1] "Nombre de Villes:"   "5"               "Time (s)="           "0.00399494171142578"
[1] "Nombre de Villes:"   "6"               "Time (s)="           "0.00300312042236328"
[1] "Nombre de Villes:"   "7"               "Time (s)="           "0.00199604034423828"
[1] "Nombre de Villes:"   "8"               "Time (s)="           "0.00299787521362305"
[1] "Nombre de Villes:"   "9"               "Time (s)="           "0.00399899482727051"
[1] "Nombre de Villes:"  "10"               "Time (s)="          "0.0159909725189209"
[1] "Nombre de Villes:" "11"               "Time (s)="         "0.129922866821289"
[1] "Nombre de Villes:" "12"               "Time (s)="         "1.44419407844543"
[1] "Nombre de Villes:" "13"               "Time (s)="         "18.9941418170929"
[1] "Nombre de Villes:" "14"               "Time (min)="         "4.72061628500621"
```

7. Performances : TSP_DP_CPP

```
for (n in 5:14){temps_cpp(TSP_DP_CPP,n)}

>>>
[1] "Nombre de Villes:"   "5"                 "Time (s)="           "0.00400090217590332"
[1] "Nombre de Villes:"  "6"                "Time (s)="          "0.0039980411529541"
[1] "Nombre de Villes:"  "7"                "Time (s)="          "0.0039980411529541"
[1] "Nombre de Villes:"   "8"                 "Time (s)="           "0.00399684906005859"
[1] "Nombre de Villes:"   "9"                 "Time (s)="           "0.00699710845947266"
[1] "Nombre de Villes:"  "10"                "Time (s)="          "0.0079948902130127"
[1] "Nombre de Villes:"  "11"                "Time (s)="          "0.0269849300384521"
[1] "Nombre de Villes:"  "12"                "Time (s)="          "0.0109939575195312"
[1] "Nombre de Villes:"  "13"                "Time (s)="          "0.0339810848236084"
[1] "Nombre de Villes:" "14"               "Time (s)="         "0.135921001434326"
[1] "Nombre de Villes:" "15"               "Time (s)="         "0.242860078811646"
[1] "Nombre de Villes:" "16"               "Time (s)="         "0.466732025146484"
[1] "Nombre de Villes:" "17"               "Time (s)="         "1.08937692642212"
[1] "Nombre de Villes:" "18"               "Time (s)="         "2.39562606811523"
[1] "Nombre de Villes:" "19"               "Time (s)="         "8.86592602729797"
[1] "Nombre de Villes:" "20"               "Time (s)="         "19.0231111049652"
[1] "Nombre de Villes:" "21"               "Time (s)="         "46.5373721122742"
[1] "Nombre de Villes:" "22"               "Time (min)="         "1.86758156617482"
[1] "Nombre de Villes:" "23"               "Time (min)="         "3.07009353240331"
```

References

[1] Rémi Watrigant. Approximation et complexité paramétrée de problèmes d'optimisation dans les graphes : partitions et sous-graphes.

[2] Sanjeeb Dash David S. Johnson David L. Applegate, William J. Cook. A practical guide to discrete optimization.