Evaluation des algorithmes heuristiques pour le voyageur de commerce

M2 Data Science Algorithmique

Vincent Runge

vendredi 14 mars 2025

Table des matières

1 Description du problème et objectif **1**

2 L'algorithme naïf du plus proche voisin **2**

3 Les algorithmes d'insertion **3**

3.1 L'algorithme d'insertion *cheapest* ("le moins cher") 4

3.2 L'algorithme d'insertion *nearest* ("le plus proche") 4

3.3 L'algorithme d'insertion *farthest* ("le plus éloigné") 5

4 Comparaison des performances **6**

4.1 Pour les différents algorithmes heuristiques 6

4.2 Pour une distribution normale des villes 7

5 Temps de calcul **8**

6 Algorithme *Branch and Bound* and Programmation Dynamique **12**

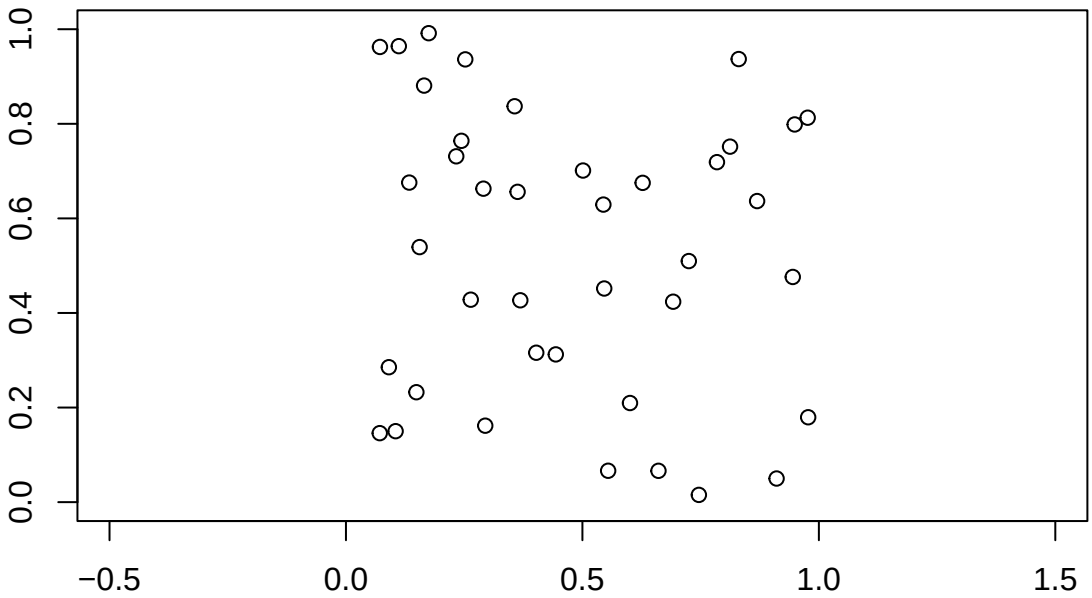
7 Amélioration de tour par 2-opt et 3-opt **13**

---

1 Description du problème et objectif

On tire de manière aléatoire dans le carré unité selon une loi uniforme $\mathcal{U}(0,1) \times \mathcal{U}(0,1)$ un nombre `n` de villes. On donne ici un exemple avec 40 villes :

```r
n <- 40
villes <- matrix(runif(2*n), n, 2)
```



Notre premier objectif est de contruire un "plus court chemin" par un **algorithme heuristique**. On comparera différentes **méthodes d'insertion** et on analysera leur **temps de calcul** numériquement.

Notre objectif (TP) :

- comparer les performances en temps des algorithmes
- évaluer la distance à la solution optimale
- pour cela coder les algorithmes exacts de *branch and bound* et *Held Karp*.

On note $c(i, j)$ le coût pour passer de la ville i à la ville j. Notre objectif est de trouver la permutation des indices $(1, ..., n)$ notée $(v_1, ..., v_n)$ qui minimisera la longueur du tour :

$$\sum_{i=1}^{n} c(v_i, v_{i+1})$$

avec $v_{n+1} = v_1$. Remarquez bien qu'une permutation contient une et une seule fois chaque indice de sorte que le tour est complet et passe bien par chaque ville une et une seule fois. Le coût $c(i, j)$ sera dans ce document égal à la distance euclidienne.

2 L'algorithme naïf du plus proche voisin

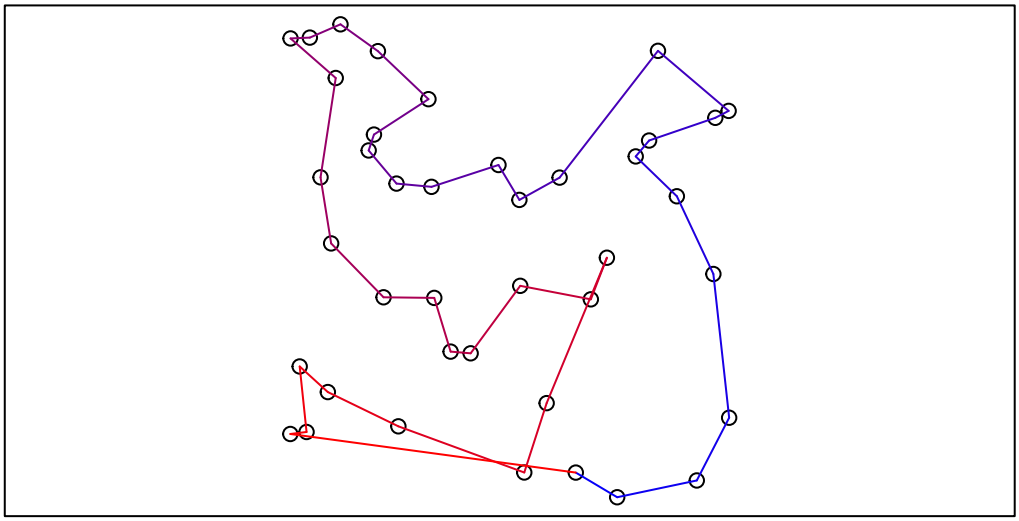
C'est la méthode la plus simple. Elle consiste à partir d'une ville i et de contruire le chemin de proche en proche en ajoutant en bout de chemin la ville la plus proche parmi les villes non explorées. Quand toutes les villes sont explorées on revient à la première ville pour fermer le tour.

On peut répéter la procédure pour chaque ville de départ (on exécute ainsi n fois cette méthode) pour choisir le meilleur chemin parmi les n obtenus.

(librairies à installer)

```r
library(ggplot2) #ggplot
library(reshape2) #melt
library(parallel) #mclapply
```

```r
library(M2algorithmique)
res1 <- TSP_naif(villes, type = "one")
```



```
## [1] 5.808868
```

```
## Longueur = 5.808868
```

Exercice : pour le problème euclidien du voyageur de commerce (inégalité triangulaire respectée), le tour optimal ne peut pas contenir de croisement.

En effet, supposons que le tour contienne un croisement. Cela signifie qu'il existe quatre villes A,B,C,D telles que les arêtes (A,C) et (B,D) se croisent. On échange les arêtes croisées : connecter A à B puis C à D. Cela supprime le croisement.

Si AC et BD se coupent en O. On a :

$$d(A,B) \leq d(A,O) + d(O,B)\,, \quad d(C,D) \leq d(C,O) + d(O,D)$$

donc

$$d(A,B) + d(C,D) \leq d(A,O) + d(O,B) + d(C,O) + d(O,D) = d(A,C) + d(B,D)$$

Dans le tour, $A => C$ et $B => D$ est remplacé par $A => B$ et $C => D$, l'entrée par A et la sortie par D sont respectés. Tout autre permutation des lettres est possible et mène au même résultat

---

3 Les algorithmes d'insertion

les algorithmes d'insertion consistent à insérer les villes les unes après les autres **dans un tour partiel** (contenant qu'un sous-ensemble des villes) partant d'une ou deux villes de départ.

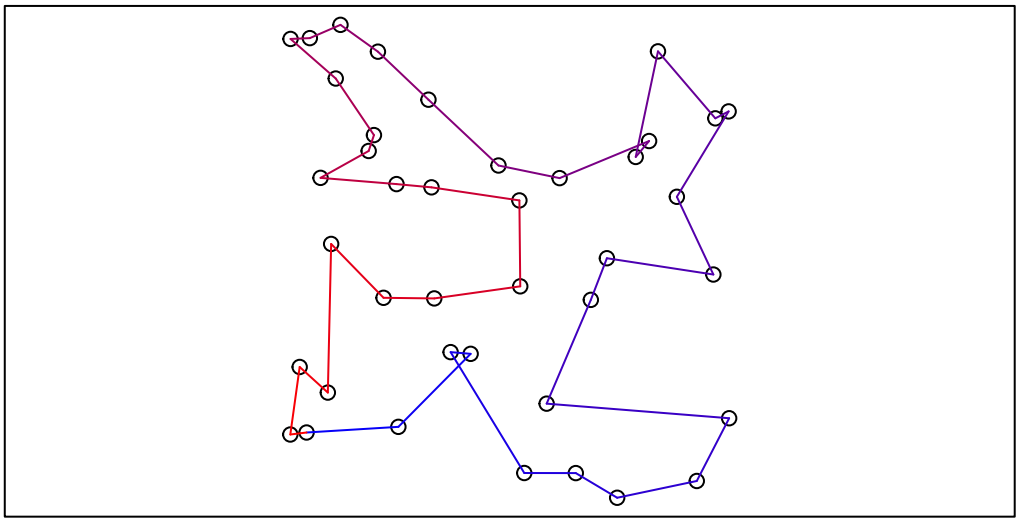
3.1 L'algorithme d'insertion *cheapest* ("le moins cher")

Pour un tour partiel déjà constitué on cherche l'arrête (le couple de villes) (i, j) et la ville encore non incluse k qui minimise la quantité

$$c(i, k) + c(k, j) - c(i, j)$$

C'est ainsi l'insertion la moins coûteuse. On pourra aussi répéter l'algorithme pour chacune des villes de départ.

```
res2 <- TSP_cheapest(villes, type = "one")
```



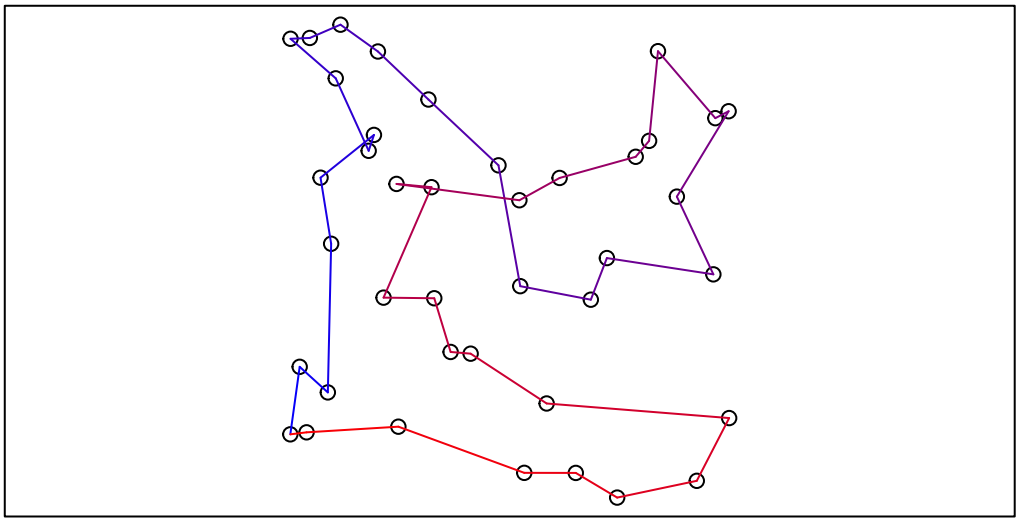
```
## [1] 5.921506
```

```
## Longueur = 5.921506
```

3.2 L'algorithme d'insertion *nearest* ("le plus proche")

Pour un tour partiel déjà constitué on cherche la ville i et la ville encore non incluse k qui minimise la quantité $c(i, k)$: c'est la ville la plus proche du tour. Une fois trouvée on insert cette ville à sa position optimale en trouvant l'arrête (i, j) qui minimise $c(i, k) + c(k, j) - c(i, j)$ C'est ainsi l'insertion la plus proche. On pourra aussi répéter l'algorithme pour chacune des villes de départ.

```
res3 <- TSP_nearest(villes)
```



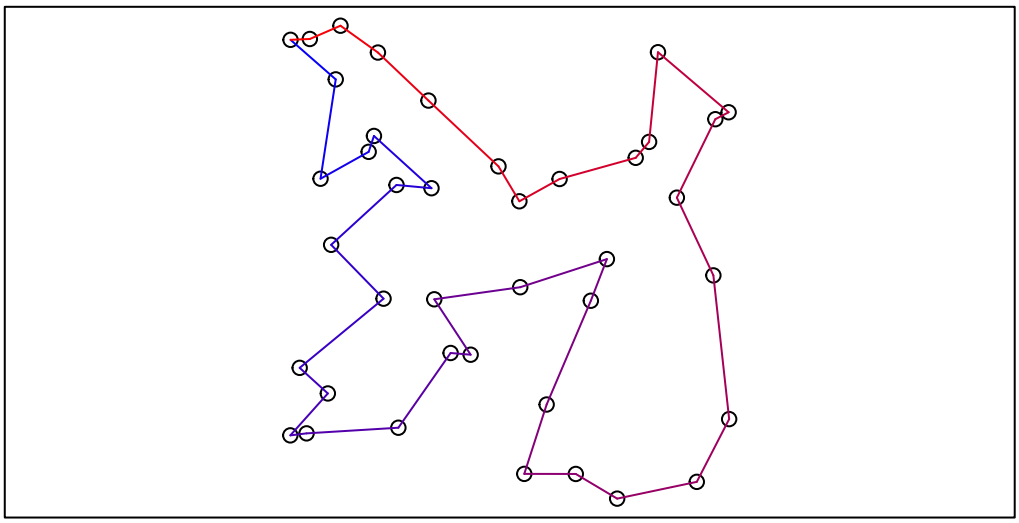
```
## [1] 5.89975
```

```
## Longueur = 5.89975
```

3.3 L'algorithme d'insertion *farthest* ("le plus éloigné")

Pour un tour partiel déjà constitué on cherche pour chaque ville non encore incluse k, la ville i du tour la plus proche. On obtient des distances $c(i,k)$ avec autant de couples (i,k) qu'il y a de villes non incluses. On sélectionne le plus grande de ces distances et la ville k qui lui est associée. On insère cette ville k à sa position optimale selon le critère habituel (min de $c(i,k)+c(k,j)-c(i,j)$).

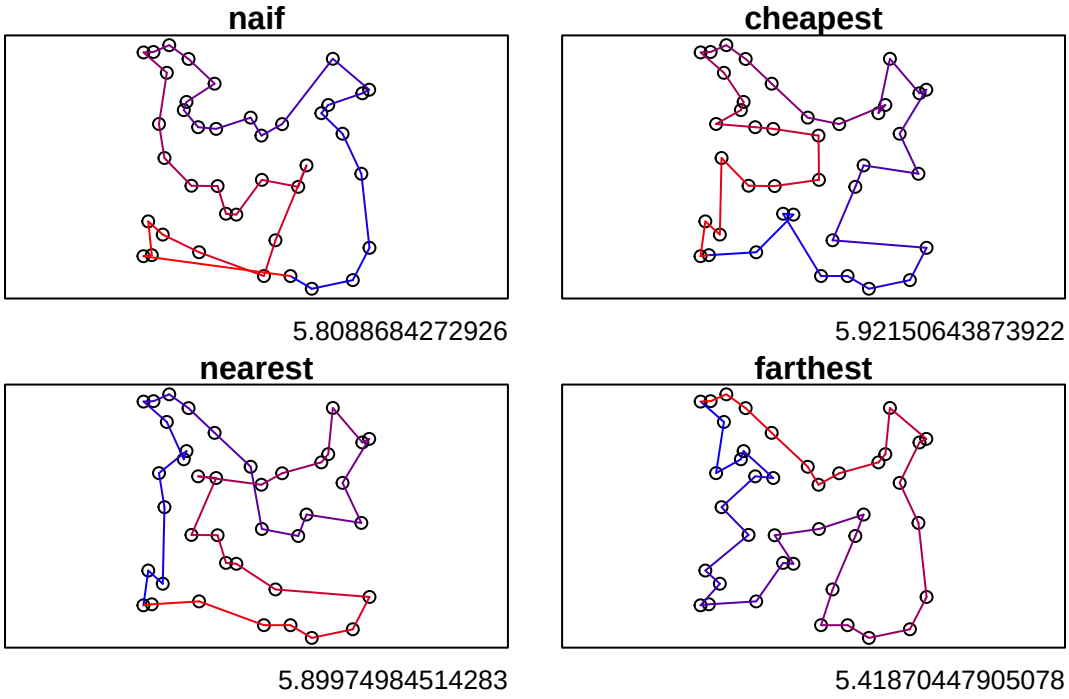
```
res4 <- TSP_farthest(villes)
```



```
## [1] 5.418704
```

```
## Longueur = 5.418704
```

Affichés tous ensemble :



Il est possible dans ce cadre eucliden d'obtenir une bornes sur la longueur du tour. Ces algorithmes heuristiques sont donc des algorithmes d'approximation (sauf peut-être pour *farthest*) !

$$algo(cheapest) \leq 2\, algo(opt)$$

$$algo(nearest) \leq 2\, algo(opt)$$

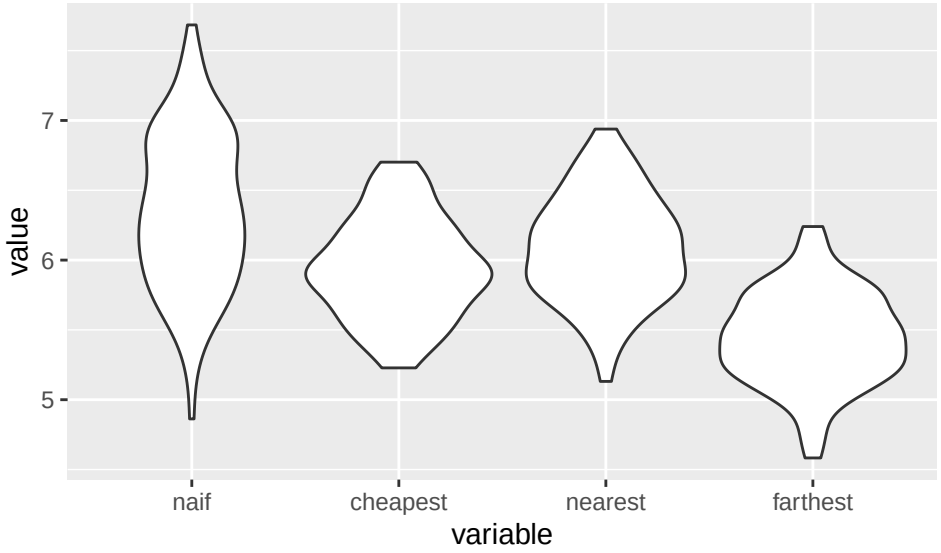
$$algo(farthest) \leq (\lceil log_2(n) \rceil + 1)\, algo(opt)$$

4 Comparaison des performances

4.1 Pour les différents algorithmes heuristiques

On répète 100 fois les 4 algorithmes sur des données générées par $\mathcal{U}[0,1] \times \mathcal{U}[0,1]$

```
## No id variables; using all as measure variables
```



Rang moyen :

```
##     naif cheapest  nearest farthest
##     3.46     2.51     2.99     1.04
```

4.2 Pour une distribution normale des villes

On répète 100 fois les 4 algorithmes sur des données normales générées par $\mathcal{N}(0,1) \times \mathcal{N}(0,1)$

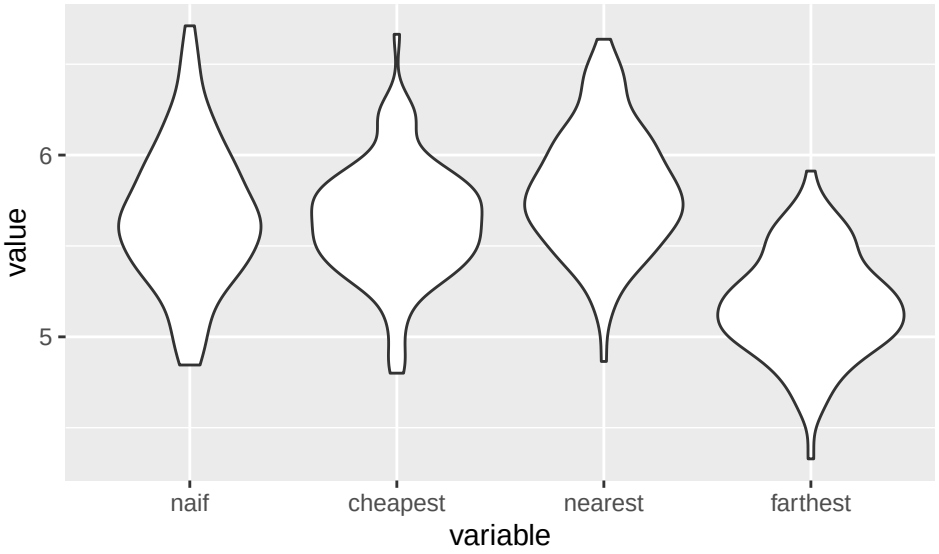
```
## No id variables; using all as measure variables
```

Rang moyen :

```
##     naif cheapest  nearest farthest
##     3.31     2.59     3.04     1.06
```

Comment évoluent ces résultats si on répète chaque algorithme pour les n initialisations possibles ? Avec la distribution uniforme des villes on obtient :

```
## No id variables; using all as measure variables
```



Rang moyen :

```
##     naif cheapest  nearest farthest
##     2.85     2.68     3.47     1.00
```

5 Temps de calcul

On étudie ici le temps la complexité des algorithmes en fonction du nombre n de villes.

On définit une fonction de type `one.simu` qui simule une seule expérience pour un choix de ville.

```
one.simu_time_TSP <- function(i, data, algo = "naif", type = "one")
{
  if(algo == "naif")
  {
    start_time <- Sys.time()
    TSP_naif(data, type = type)
    end_time  <- Sys.time()
  }
  if(algo == "cheapest")
  {
    start_time <- Sys.time()
    TSP_cheapest(data, type = type)
    end_time  <- Sys.time()
  }
  if(algo == "nearest")
  {
    start_time <- Sys.time()
    TSP_nearest(data, type = type)
```

```
    end_time  <- Sys.time()
  }
  if(algo == "farthest")
  {
    start_time <- Sys.time()
    TSP_farthest(data, type = type)
    end_time  <- Sys.time()
  }
  return(unclass(end_time - start_time)[1])
}
```

On construit un vecteur contenant **des tailles croissante de nombre de villes** selon une échelle logarithmique.

```
my_n_vector_LOG <- seq(from = log(10), to = log(100), by = log(10)/40)
my_n_vector <- round(exp(my_n_vector_LOG))
my_n_vector
```

```
##  [1]  10  11  11  12  13  13  14  15  16  17  18  19  20  21  22  24  25  27  28
## [20]  30  32  33  35  38  40  42  45  47  50  53  56  60  63  67  71  75  79  84
## [39]  89  94 100
```

```
diff(log(my_n_vector))
```

```
##  [1] 0.09531018 0.00000000 0.08701138 0.08004271 0.00000000 0.07410797
##  [7] 0.06899287 0.06453852 0.06062462 0.05715841 0.05406722 0.05129329
## [13] 0.04879016 0.04652002 0.08701138 0.04082199 0.07696104 0.03636764
## [19] 0.06899287 0.06453852 0.03077166 0.05884050 0.08223810 0.05129329
## [25] 0.04879016 0.06899287 0.04348511 0.06187540 0.05826891 0.05505978
## [31] 0.06899287 0.04879016 0.06155789 0.05798726 0.05480824 0.05195974
## [37] 0.06136895 0.05781957 0.05465841 0.06187540
```

On voit que `diff(log())` est à peu près constant. Ce n'est pas constant à cause de l'arrondi avec `round`. On construit un data frame qui contiendra les résultats :

```
p <- 50 ### répétition
df <- data.frame(matrix( nrow = 4 * length(my_n_vector), ncol = 2 + p))
colnames(df) <- c("type", "n", 1:p)
dim(df)
```

```
## [1] 164  52
```

On lance la simulation sur plusieurs coeurs.

```
nbCores <- 8
j <- 1

for(n in my_n_vector)
{
  #print(n)
  liste1 <- mclapply(1:p, FUN = one.simu_time_TSP,
```

```
                      data = matrix(runif(2*n), n, 2),
                     algo = "naif",
                     mc.cores = nbCores)

  liste2 <- mclapply(1:p, FUN = one.simu_time_TSP,
                     data = matrix(runif(2*n), n, 2),
                     algo = "cheapest",
                     mc.cores = nbCores)

  liste3 <- mclapply(1:p, FUN = one.simu_time_TSP,
                     data = matrix(runif(2*n), n, 2),
                     algo = "nearest",
                     mc.cores = nbCores)

  liste4 <- mclapply(1:p, FUN = one.simu_time_TSP,
                     data = matrix(runif(2*n), n, 2),
                     algo = "farthest",
                     mc.cores = nbCores)

  df[j ,] <- c("naif", n, do.call(cbind, liste1))
  df[j+1 ,] <- c("cheapest", n, do.call(cbind, liste2))
  df[j+2 ,] <- c("nearest", n, do.call(cbind, liste3))
  df[j+3 ,] <- c("farthest", n, do.call(cbind, liste4))
  j <- j + 4
}

df <- melt(df, id.vars = c("type","n"))
```

tranformations techniques :

```
data_summary <- function(data, varname, groupnames)
{
  require(plyr)
  summary_func <- function(x, col)
  {
    c(mean = mean(x[[col]], na.rm=TRUE),
      q1 = quantile(x[[col]], 0.025), q3 = quantile(x[[col]], 0.975))
  }
  data_sum<-ddply(data, groupnames, .fun=summary_func,
                  varname)
  data_sum <- rename(data_sum, c("mean" = varname))
  return(data_sum)
}

df2 <- df
df2[,2] <- as.double(df[,2])
df2[,3] <- as.double(df[,3])
df2[,4] <- as.double(df[,4])
summary(df2)
```

```
##      type                  n             variable        value
##  Length:8200        Min.   : 10.00   Min.   : 1.0   Min.   :0.0000529
##  Class :character   1st Qu.: 18.00   1st Qu.:13.0   1st Qu.:0.0008280
```

```
##  Mode  :character   Median : 32.00   Median :25.5   Median :0.0043460
##                     Mean   : 39.49   Mean   :25.5   Mean   :0.0316682
##                     3rd Qu.: 56.00   3rd Qu.:38.0   3rd Qu.:0.0322667
##                     Max.   :100.00   Max.   :50.0   Max.   :0.5555508
```

```
df_new <- data_summary(df2, varname="value",
                           groupnames=c("type","n"))
```

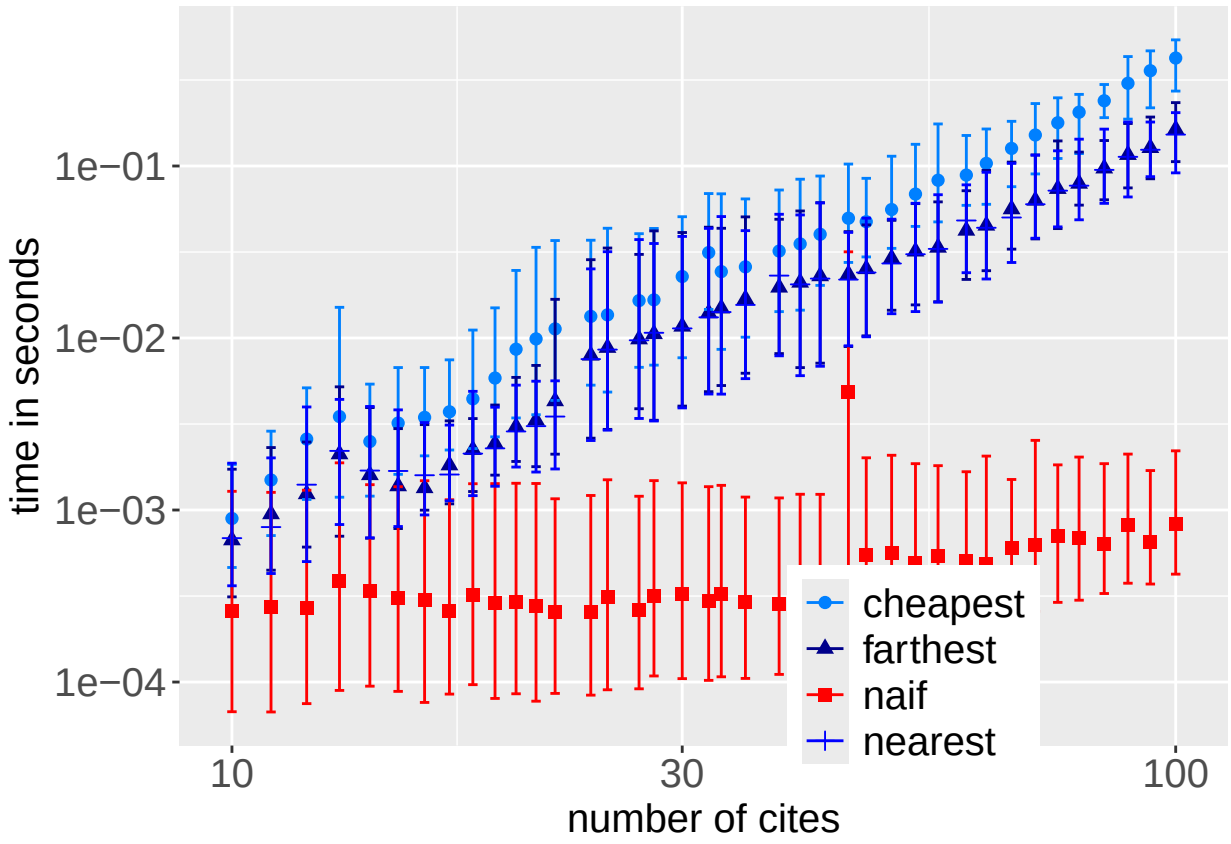
```
## Loading required package: plyr
```

```
theMin <- min(df_new[,3:5],df_new[,3:5])
theMax <- max(df_new[,3:5],df_new[,3:5])
```

On trace différentes courbes.

```
ggplot(df_new, aes(x = n, y = value, col=type)) +  scale_x_log10()+
  scale_y_log10(limits = c(theMin, theMax))  +
  labs(y = "time in seconds") +  labs(x = "number of cites") +
  geom_point(size = 2, aes(shape = type)) +
  geom_errorbar(aes(ymin=`q1.2.5%`, ymax=`q3.97.5%`), width=.01) +
  scale_colour_manual(values = c("cheapest" = "#0080FF",
                                 "farthest" = " dark blue", "nearest" = "blue", "naif" = "red")) +
  theme(axis.text.x = element_text(size=15),
        axis.text.y = element_text(size=15),
        legend.text=element_text(size=15),
        axis.title.x=element_text(size=15),
        axis.title.y=element_text(size=15),
        legend.position = c(0.7, 0.1),
        legend.title = element_blank())
```

```
## Warning: A numeric ‘legend.position‘ argument in ‘theme()‘ was deprecated in ggplot2
## 3.5.0.
## i Please use the ‘legend.position.inside‘ argument of ‘theme()‘ instead.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```



On calcule les valeurs des coefficients directeurs.

Pour naif :

```
## (Intercept)      log(n)
##  -9.6181451   0.5213248
```

Pour cheapest :

```
## (Intercept)      log(n)
##  -12.301360    2.452108
```

Pour nearest :

```
## (Intercept)      log(n)
##  -12.504142    2.298085
```

Pour farthest :

```
## (Intercept)      log(n)
##  -12.555933    2.318949
```

6 Algorithme *Branch and Bound* and Programmation Dynamique

— Ajouter la fonction `B_and_B`

— Ajouter la fonction `Held_Karp` (programmation dynamique)
— Evaluer le coefficient d'approximation des méthodes dans le cas d'une répartition uniforme et normale des villes

7 Amélioration de tour par 2-opt et 3-opt

EXERCIC Bonus :

— Ajouter les fonctions `opt2` et `opt3` à coder
— Evaluer l'amélioration apportée en terme de distance