Evaluation des algorithmes heuristiques pour le voyageur de commerce avec les méthodes d'insertion

M2 Data Science Algorithmique
Exemple de projet

Vincent Runge

jeudi 27 octobre 2022

Table des matières

Librairies à installer :

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

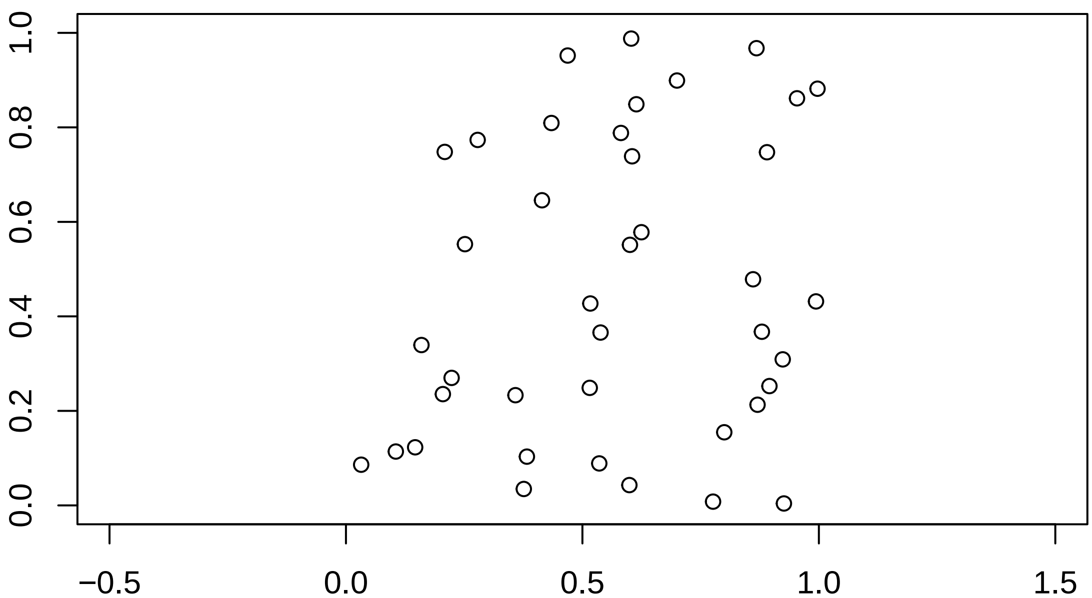
La librairie de ce travail est disponible sur github :

```r
#devtools::install_github("vrunge/TSP")
library(TSP)
```

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

Nos objectifs :

- comparer les performances en temps des différents algorithmes
- évaluer la distance à la solution optimale
- pour cela coder un algorithme exact (par exemple *branch and bound*).

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 $v_{n+1} = v_1$ (pour revenir à la ville de départ). 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.

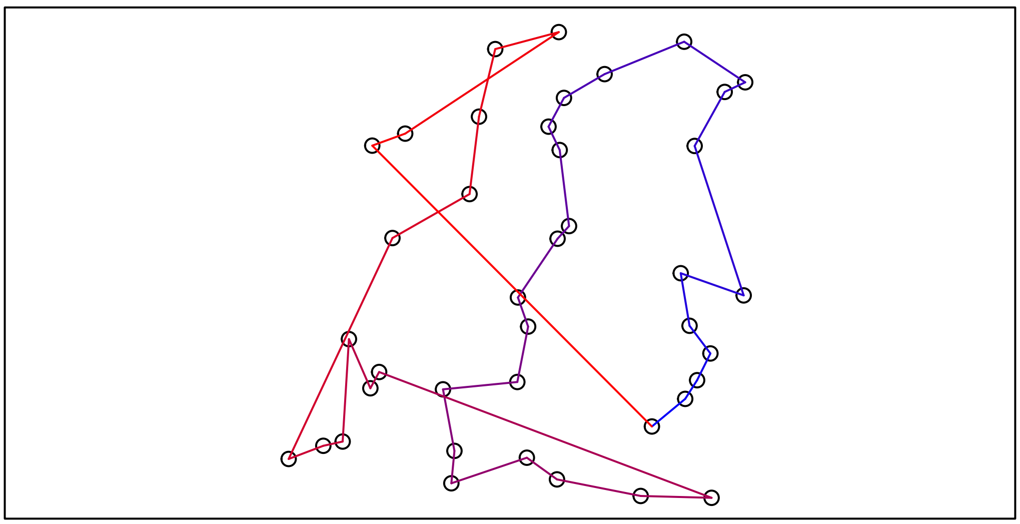
L'algorithme naïf du plus proche voisin

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.

Exemple avec une seule ville de départ :

```
res1 <- NN_TSP(villes)
plot(x = res1, data = villes)
```



```
(t1 <- tour_length(res1, villes))
```

```
## [1] 6.762322
```

On remarque que la fermeture du tour est assez peu optimale...

EXERCICE : pour le problème euclidien du voyageur de commerce (inégalité triangulaire respectée), le tour optimal ne peut pas contenir de croisement. **Le prouver !**

Les algorithmes d'insertion

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

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

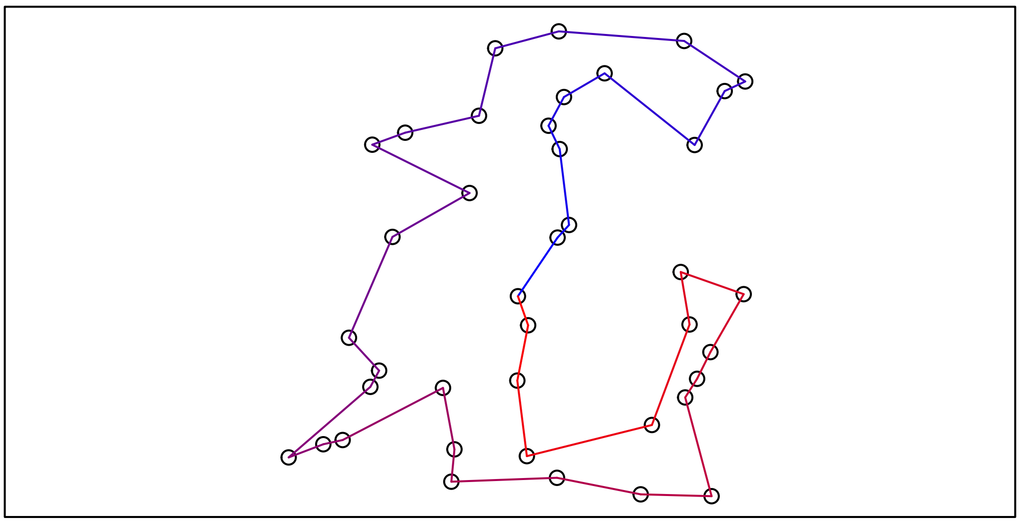
Pour un tour partiel déjà constitué on cherche l'arrête (le couple de villes) (i, j) et la ville encore non incluse k qui minimise la quantité

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

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

Un exemple avec une seule ville de départ :

```
res2 <- greedy_TSP_best(villes)
plot(x = res2, data = villes)
```



```
(t2 <- tour_length(res2, villes))
```

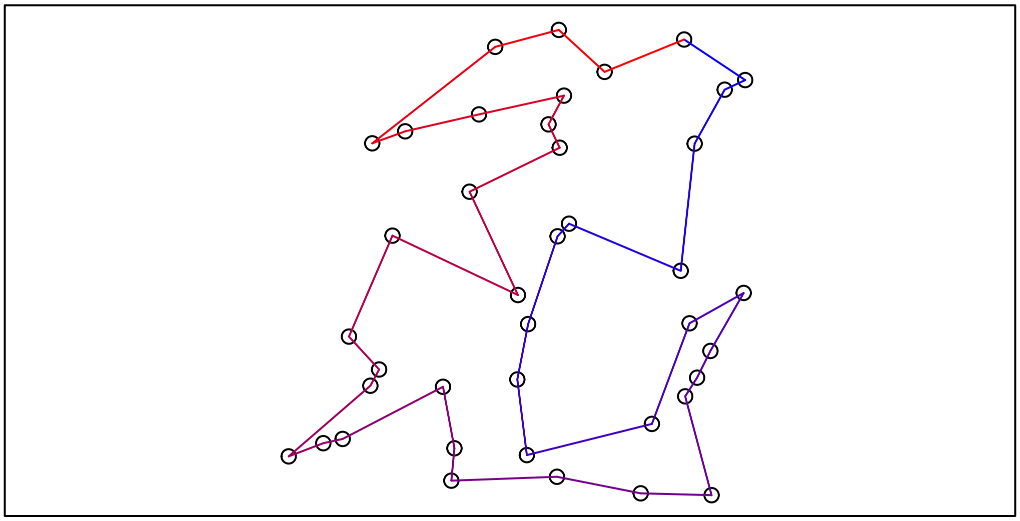
```
## [1] 5.625291
```

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.

Un exemple avec une seule ville de départ :

```
res3 <- greedy_TSP_min(villes)
plot(x = res3, data = villes)
```



```
(t3 <- tour_length(res3, villes))
```

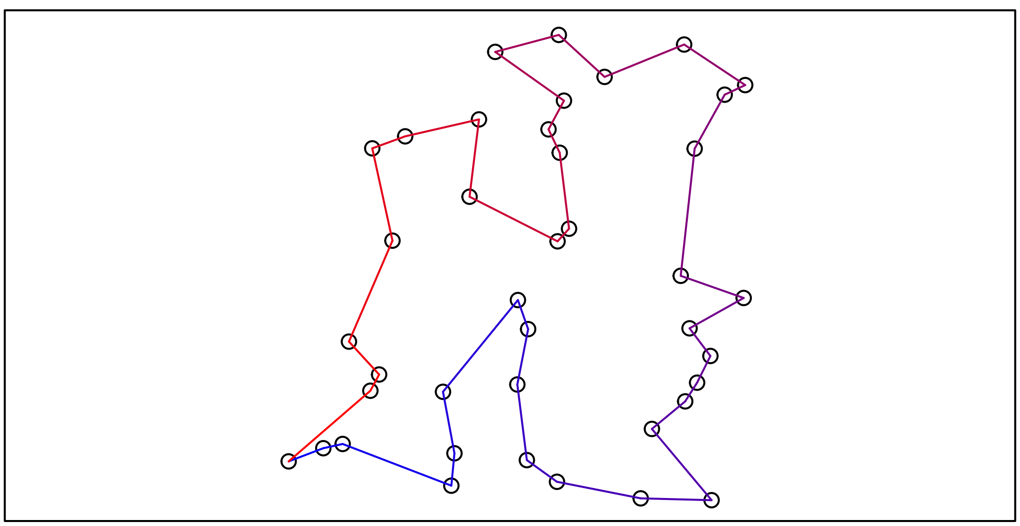
```
## [1] 6.249731
```

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

Un exemple avec une seule ville de départ :

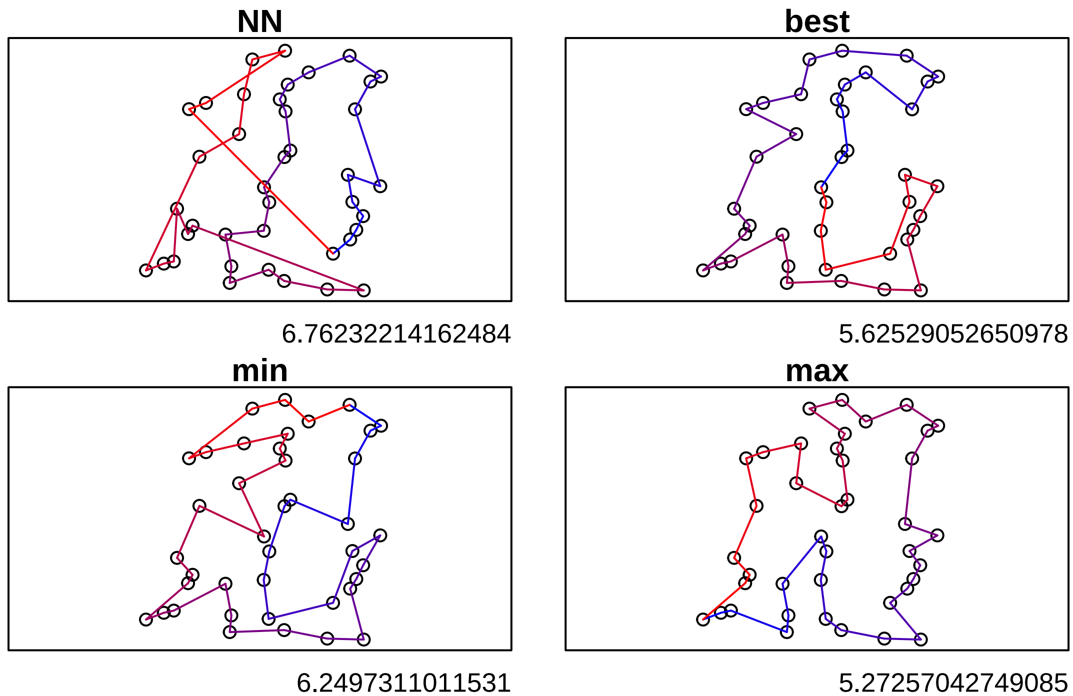
```
res4 <- greedy_TSP_max(villes)
plot(x = res4, data = villes)
```



```r
(t4 <- tour_length(res4, villes))
```

```
## [1] 5.27257
```

Affichés tous ensemble :



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

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

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

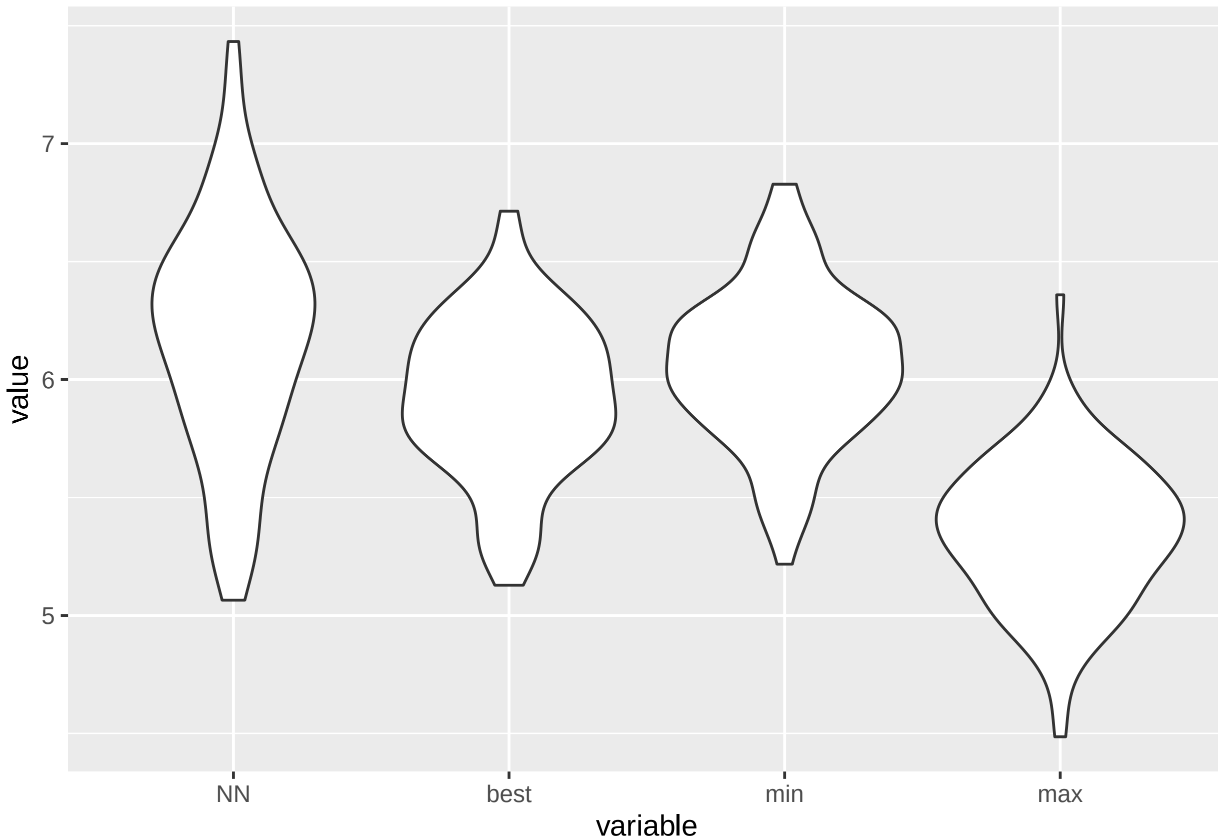
$$algo(farthest) \leq (\lceil log_2(n) \rceil + 1)\, algo(opt)$$

Comparaison des performances

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
```



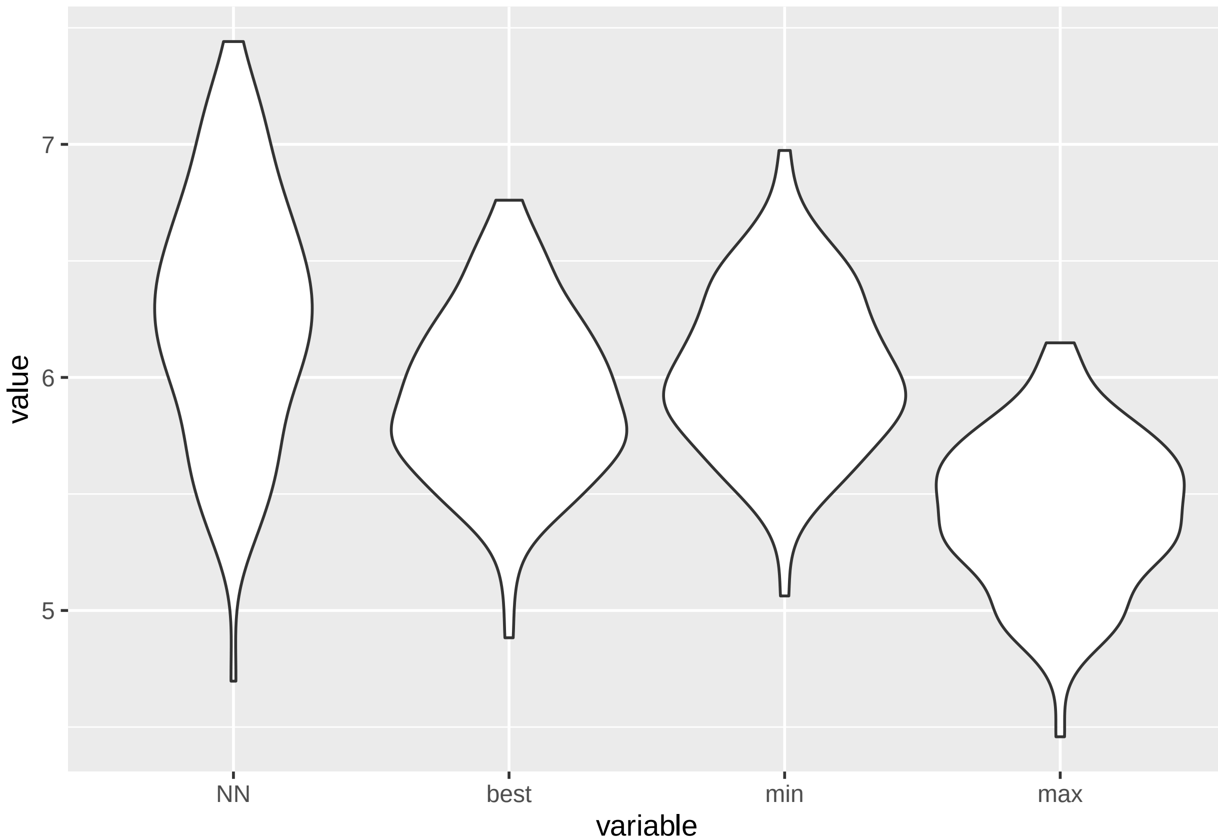
Rang moyen :

```
##   NN best  min  max
## 3.26 2.52 3.17 1.05
```

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
```



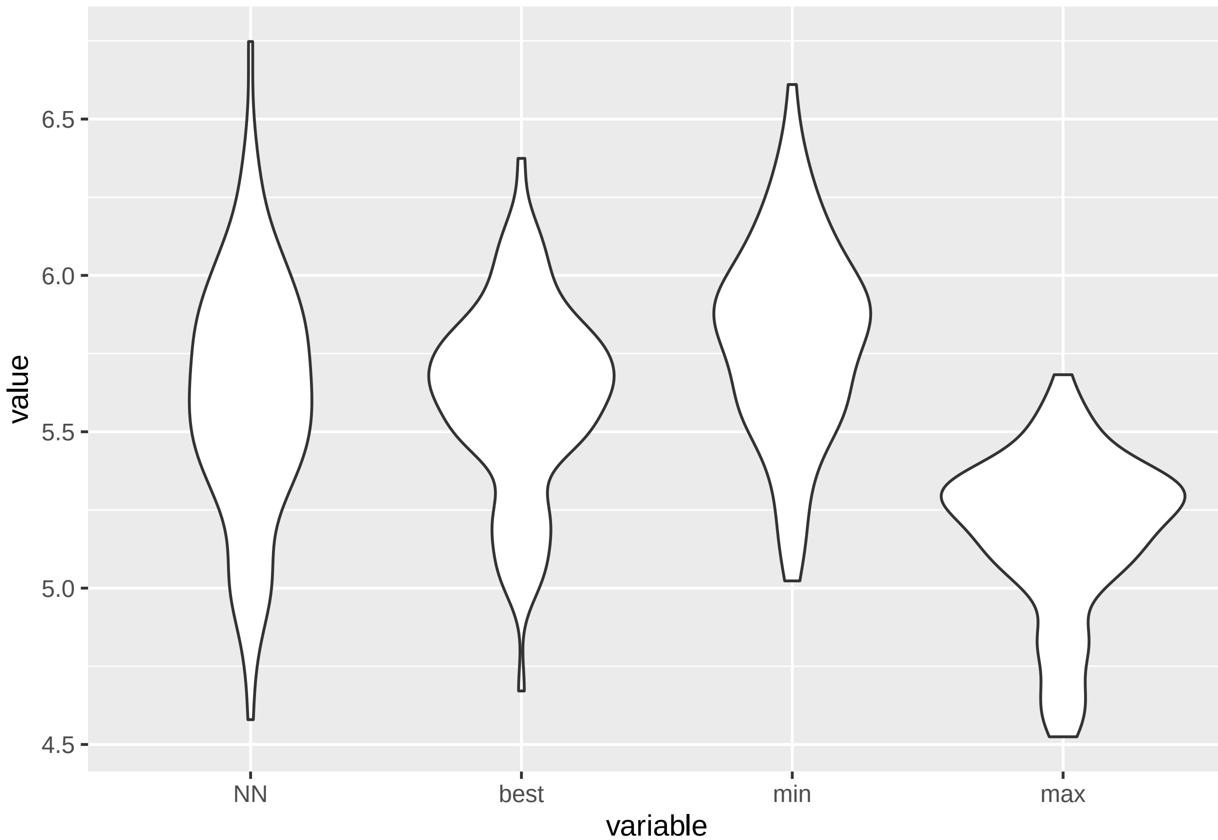
Rang moyen :

```
##   NN best  min  max
## 3.44 2.48 3.04 1.04
```

EXERCICE : Comment évoluent ces résultats si on répète chaque algorithme pour les n initialisations possibles ?

REPONSE (distribution uniforme) :

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



Rang moyen :

```
##   NN best  min  max
## 2.72 2.60 3.66 1.02
```

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 = "NN", type = "one")
{
  if(algo == "NN")
  {
    start_time <- Sys.time()
    NN_TSP(data, type = type)
    end_time  <- Sys.time()
  }
  if(algo == "best")
  {
    start_time <- Sys.time()
    greedy_TSP_best(data, type = type)
    end_time  <- Sys.time()
  }
  if(algo == "min")
```

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

On construit un vecteur de taille de ville selon une échelle logarithmique

```r
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
```

```r
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 construit un data frame qui contiendra les résultats

```r
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.

```r
nbCores <- 8
j <- 1

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

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

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

  liste3 <- mclapply(1:p, FUN = one.simu_time_TSP,
                    data = matrix(runif(2*n), n, 2),
                    algo = "min",
                    mc.cores = nbCores)
  liste4 <- mclapply(1:p, FUN = one.simu_time_TSP,
                    data = matrix(runif(2*n), n, 2),
                    algo = "max",
                    mc.cores = nbCores)

  df[j ,] <- c("NN", n, do.call(cbind, liste1))
  df[j+1 ,] <- c("best", n, do.call(cbind, liste2))
  df[j+2 ,] <- c("min", n, do.call(cbind, liste3))
  df[j+3 ,] <- c("max", 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.   :0.0001569
##  Class :character   1st Qu.: 18.00   1st Qu.:13.0   1st Qu.:0.0020363
##  Mode  :character   Median : 32.00   Median :25.5   Median :0.0088866
```

```
##                     Mean   : 39.49   Mean   :25.5   Mean   :0.0481867
##                     3rd Qu.: 56.00   3rd Qu.:38.0   3rd Qu.:0.0466209
##                     Max.   :100.00   Max.   :50.0   Max.   :0.8117151
```

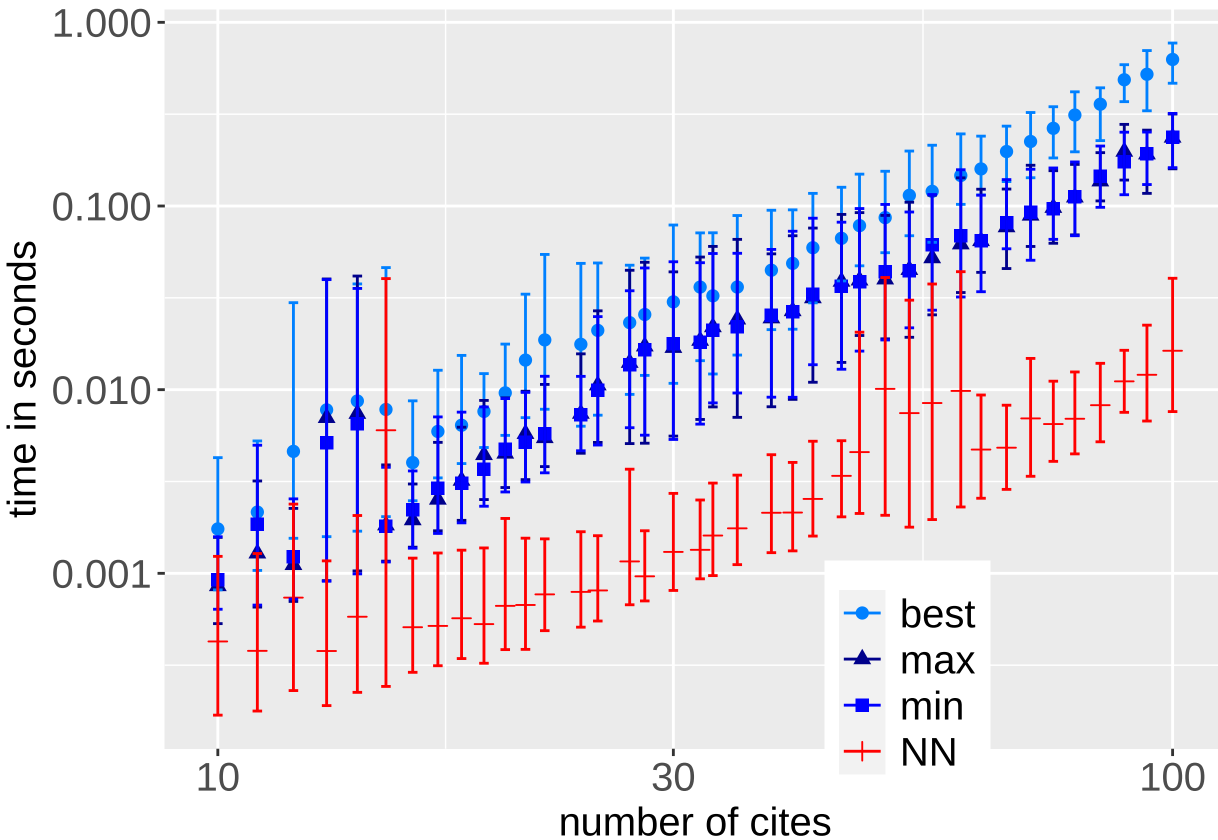
```
df_new <- data_summary(df2, varname="value",
                           groupnames=c("type","n"))
```

```
## Le chargement a nécessité le 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("best" = "#0080FF",
                                 "max" = " dark blue", "min" = "blue", "NN" = "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())
```



On calcule les valeurs des coefficients directeurs.

Pour NN :

```
R1 <- df_new[df_new$type == "NN",c(2,3)]
l1 <- lm(log(value) ~ log(n), data = R1, )
summary(l1)
```

```
##
## Call:
## lm(formula = log(value) ~ log(n), data = R1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.4894 -0.3010 -0.1811  0.1088  2.3405
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -11.7878     0.4568  -25.81  < 2e-16 ***
## log(n)        1.6001     0.1281   12.49 7.65e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5263 on 37 degrees of freedom
## Multiple R-squared:  0.8084, Adjusted R-squared:  0.8032
## F-statistic: 156.1 on 1 and 37 DF,  p-value: 7.654e-15
```

```r
l1$coefficients
```

```
## (Intercept)      log(n)
##  -11.787845    1.600138
```

Pour best :

```r
R2 <- df_new[df_new$type == "best",c(2,3)]
l2 <- lm(log(value) ~ log(n), data = R2, )
summary(l2)
```

```
##
## Call:
## lm(formula = log(value) ~ log(n), data = R2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.50370 -0.14427 -0.00081  0.08055  0.64943
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -11.59043    0.20120  -57.61   <2e-16 ***
## log(n)        2.37151    0.05642   42.04   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2318 on 37 degrees of freedom
## Multiple R-squared:  0.9795, Adjusted R-squared:  0.9789
## F-statistic:  1767 on 1 and 37 DF,  p-value: < 2.2e-16
```

```r
l2$coefficients
```

```
## (Intercept)      log(n)
##  -11.590432    2.371513
```

Pour min :

```r
R3 <- df_new[df_new$type == "min",c(2,3)]
l3 <- lm(log(value) ~ log(n), data = R3, )
summary(l3)
```

```
##
## Call:
## lm(formula = log(value) ~ log(n), data = R3)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.48062 -0.15813 -0.00262  0.10151  0.95505
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) -11.95913    0.25381  -47.12   <2e-16 ***
## log(n)        2.26161    0.07117   31.78   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2925 on 37 degrees of freedom
## Multiple R-squared:  0.9647, Adjusted R-squared:  0.9637
## F-statistic:  1010 on 1 and 37 DF,  p-value: < 2.2e-16
```

```r
l3$coefficients
```

```
## (Intercept)      log(n)
##  -11.959131    2.261613
```

Pour max :

```r
R4 <- df_new[df_new$type == "max",c(2,3)]
l4 <- lm(log(value) ~ log(n), data = R4, )
summary(l4)
```

```
##
## Call:
## lm(formula = log(value) ~ log(n), data = R4)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.54206 -0.12442 -0.07636  0.12240  1.21017
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -11.97605    0.30046  -39.86   <2e-16 ***
## log(n)        2.26425    0.08425   26.88   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3462 on 37 degrees of freedom
## Multiple R-squared:  0.9513, Adjusted R-squared:   0.95
## F-statistic: 722.3 on 1 and 37 DF,  p-value: < 2.2e-16
```

```r
l4$coefficients
```

```
## (Intercept)      log(n)
##  -11.976050    2.264249
```

Amélioration de tour par 2-opt et 3-opt

EXERCICE :

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

Algorithme *Branch and Bound*

— Ajouter la fonction `B_and_B`

— Evaluer le coefficient d'approximation des méthodes dans le cas d'une répartition uniforme et normale des villes

— Trouver pour un temps maximal donné et un nombre de villes donnés quelle est la meilleure méthode à utiliser parmis toutes celles proposées !