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

M2 Data Science Algorithmique
Exemple de projet

Vincent Runge

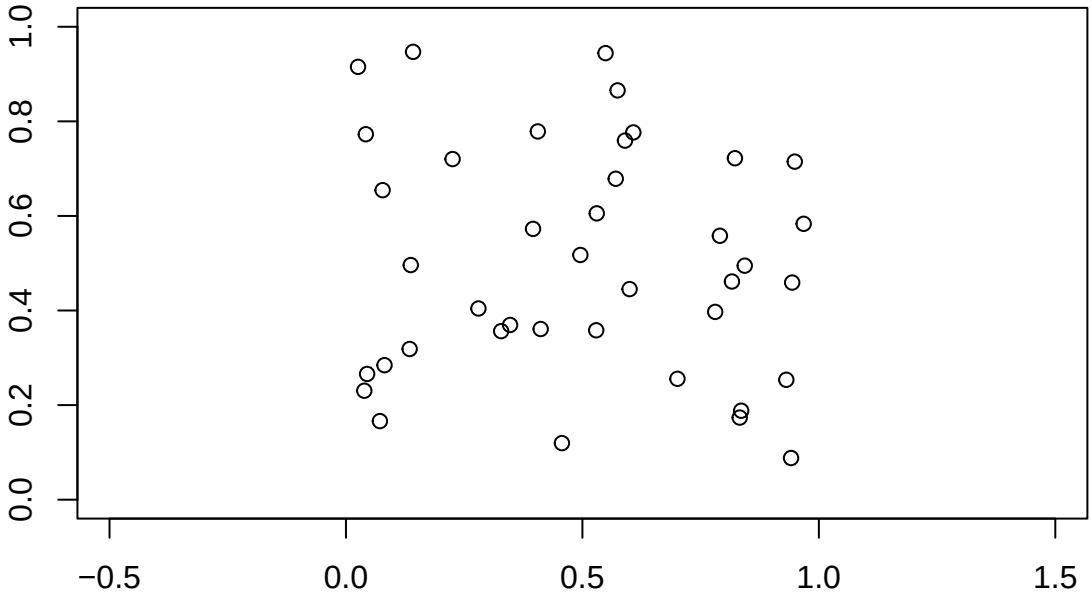
jeudi 27 octobre 2022

Table des matières

Description du problème et objectif

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

```
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.

Nos objectifs :

— comparer les performances en temps des algorithmes
— évaluer la distance à la solution optimale
— pour cela coder l'algorithme de *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 $v_{n+1} = v_1$. 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

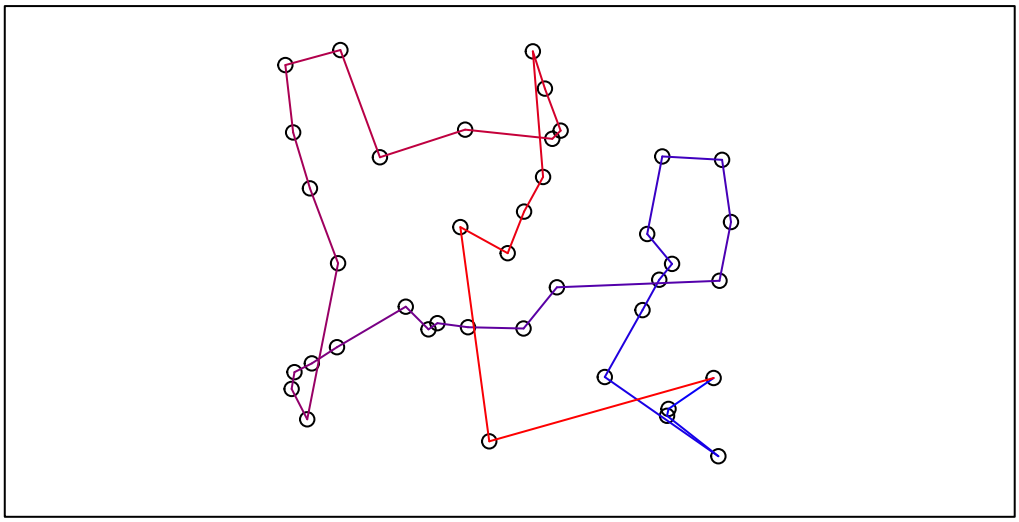
C'est la méthode la plus simple. Elle consiste à partir d'une ville i et de contruire le chemin de proche en proche en ajoutant en bout de chemin la ville la plus proche parmi les villes non explorées. Quand toutes les villes sont explorées on revient à la première ville pour fermer le tour.

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.

(librairies à installer)

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

```
library(TSP)
res1 <- NN_TSP(villes)
plot(tour = res1, data = villes)
```



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

```
## [1] 5.807853
```

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.

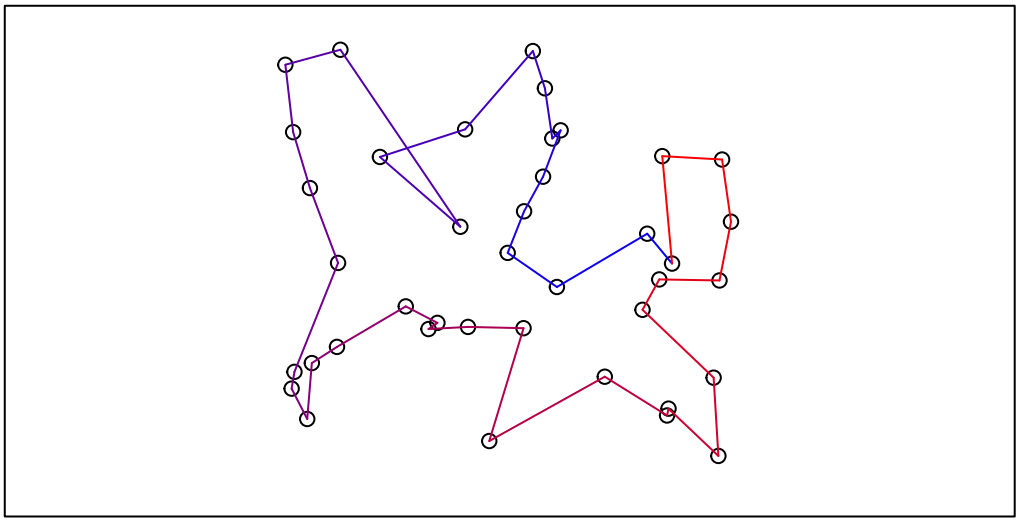
L'algorithme d'insertion *cheapest* ("le moins cher")

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.

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



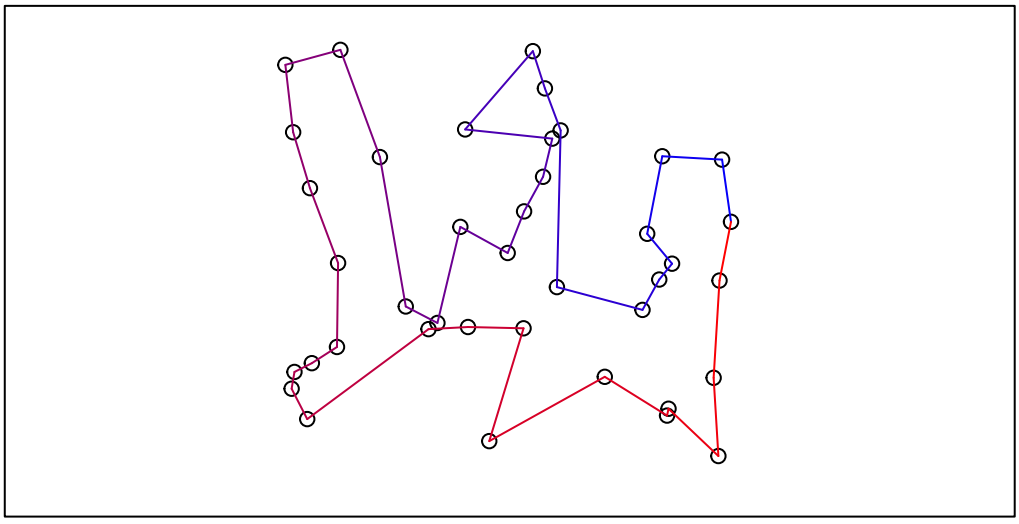
```
(t2 <- tour_length(res2, villes))
```

```
## [1] 5.610212
```

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



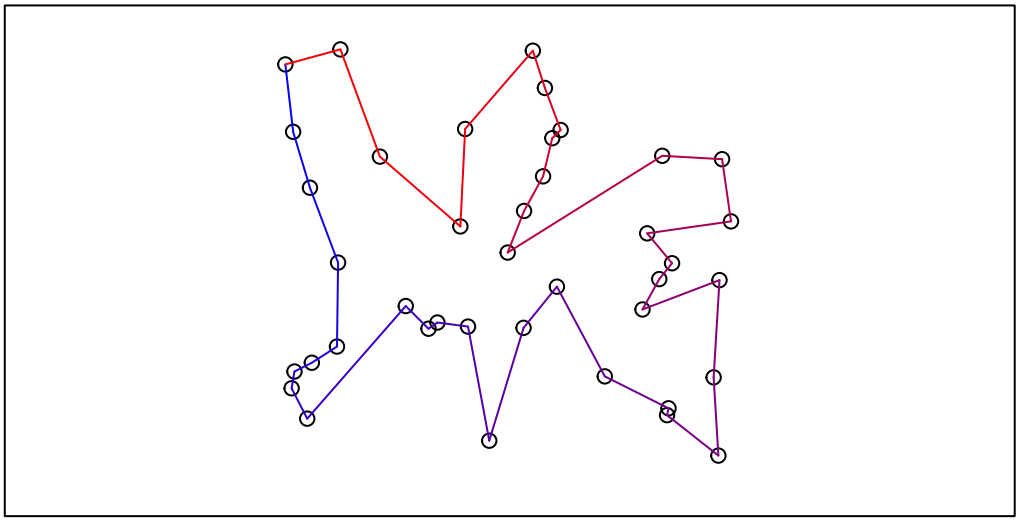
```
(t3 <- tour_length(res3, villes))
```

```
## [1] 5.757644
```

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

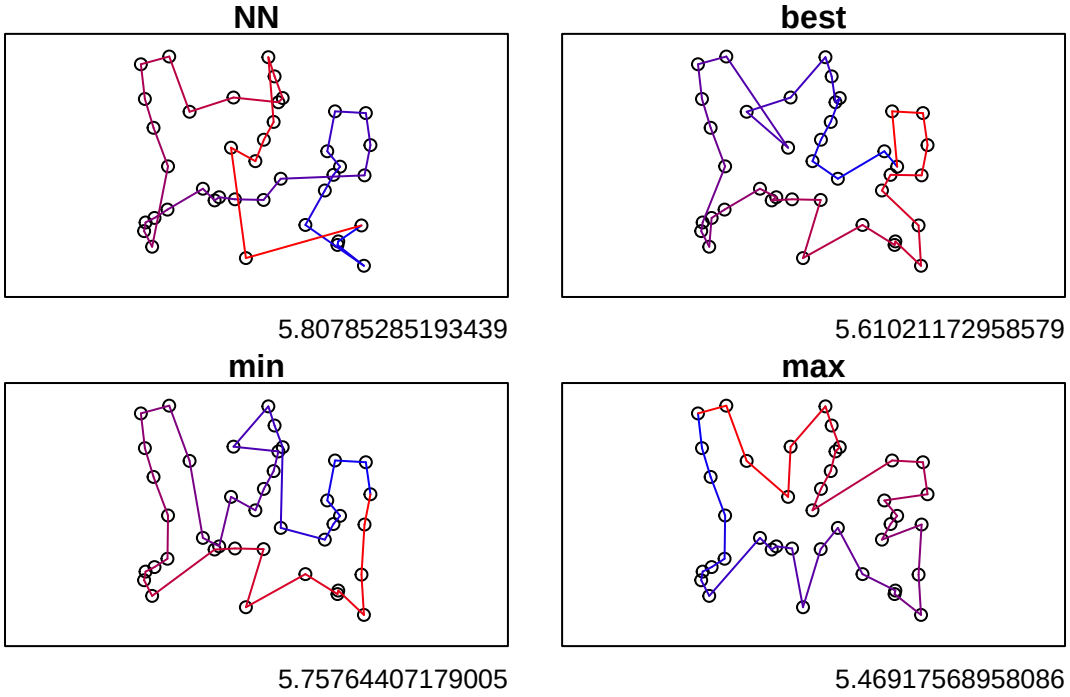
```
res4 <- greedy_TSP_max(villes)
plot(tour = res4, data = villes)
```



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

```
## [1] 5.469176
```

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*) !

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

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

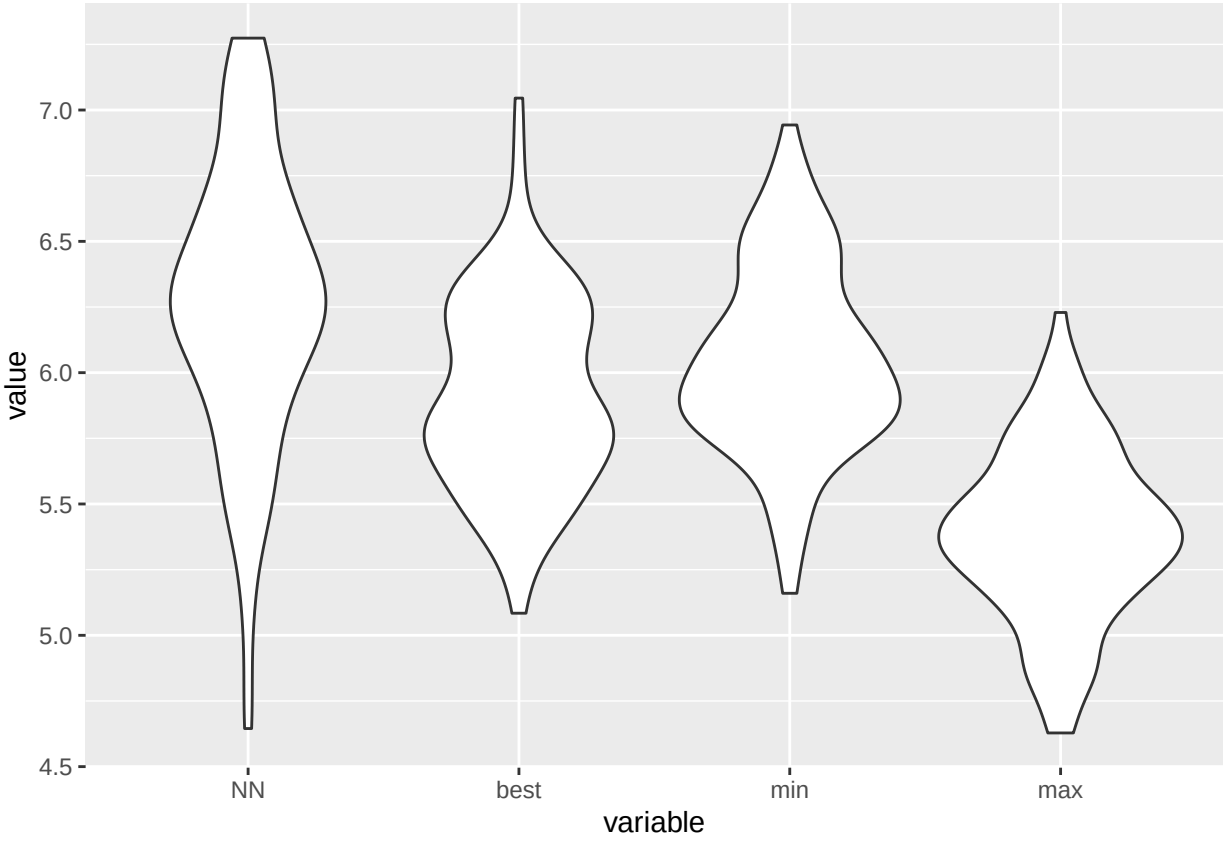
$$algo(farthest) \leq (\lceil log_2(n) \rceil + 1)\, algo(opt)$$

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
```



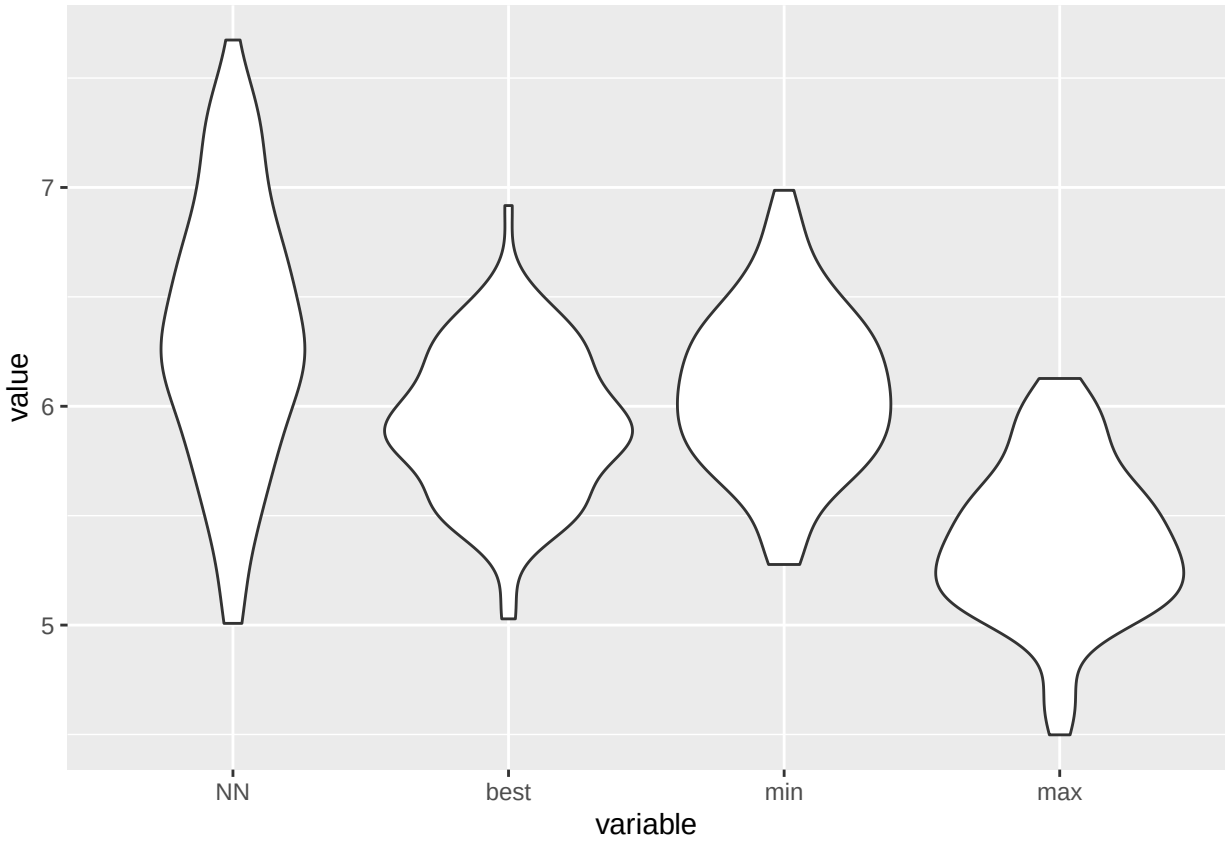
Rang moyen :

```
##   NN best  min  max
## 3.46 2.46 3.02 1.06
```

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
```



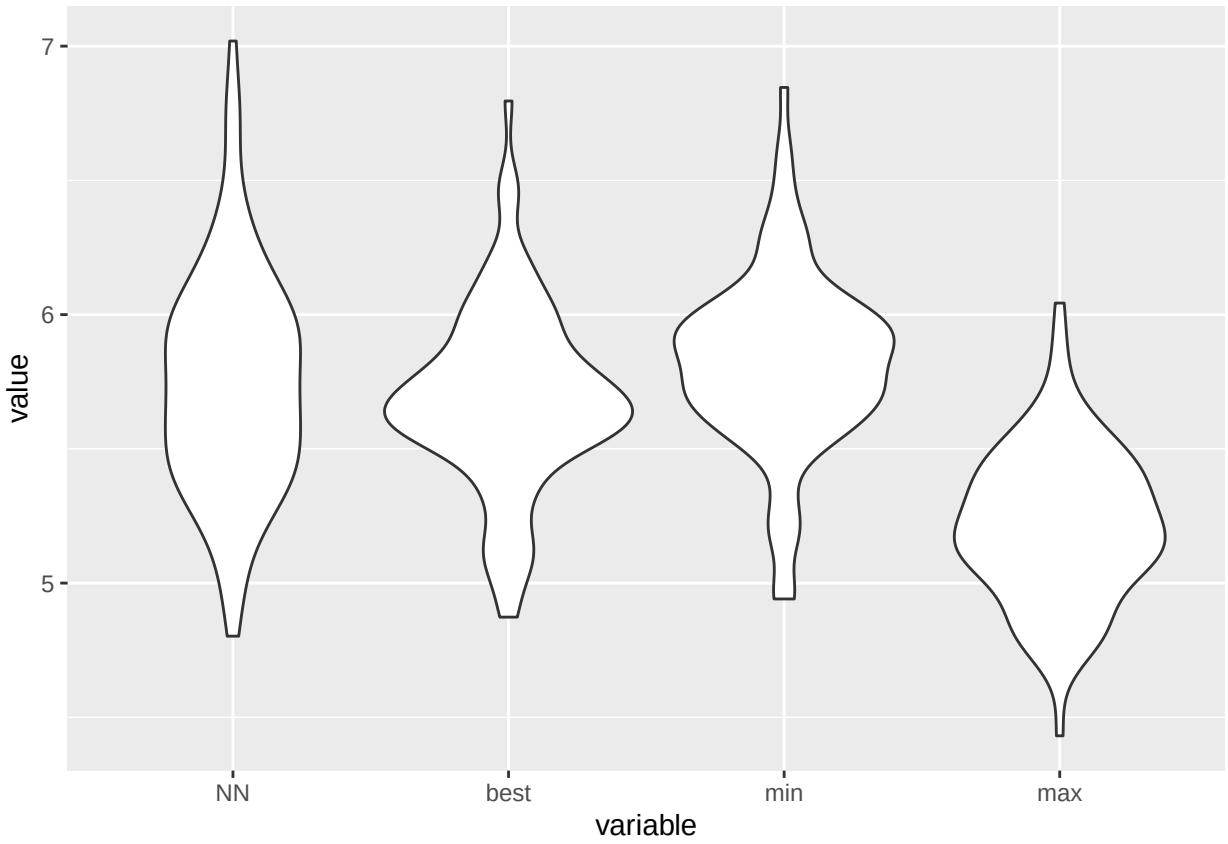
Rang moyen :

```
##    NN  best   min   max
## 3.440 2.515 3.015 1.030
```

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
## 2.930 2.665 3.390 1.015
```

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")
```

```
  {
    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

```
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 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 = "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
}
```

```
## [1] 10
## [1] 11
## [1] 11
## [1] 12
## [1] 13
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 24
## [1] 25
## [1] 27
## [1] 28
## [1] 30
## [1] 32
## [1] 33
## [1] 35
## [1] 38
## [1] 40
```

```
## [1] 42
## [1] 45
## [1] 47
## [1] 50
## [1] 53
## [1] 56
## [1] 60
## [1] 63
## [1] 67
## [1] 71
## [1] 75
## [1] 79
## [1] 84
## [1] 89
## [1] 94
## [1] 100
```

```
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.   :0.0001559
##  Class :character   1st Qu.: 18.00   1st Qu.:13.0   1st Qu.:0.0021248
##  Mode  :character   Median : 32.00   Median :25.5   Median :0.0082276
##                     Mean   : 39.49   Mean   :25.5   Mean   :0.0459139
##                     3rd Qu.: 56.00   3rd Qu.:38.0   3rd Qu.:0.0453522
##                     Max.   :100.00   Max.   :50.0   Max.   :0.7414269
```

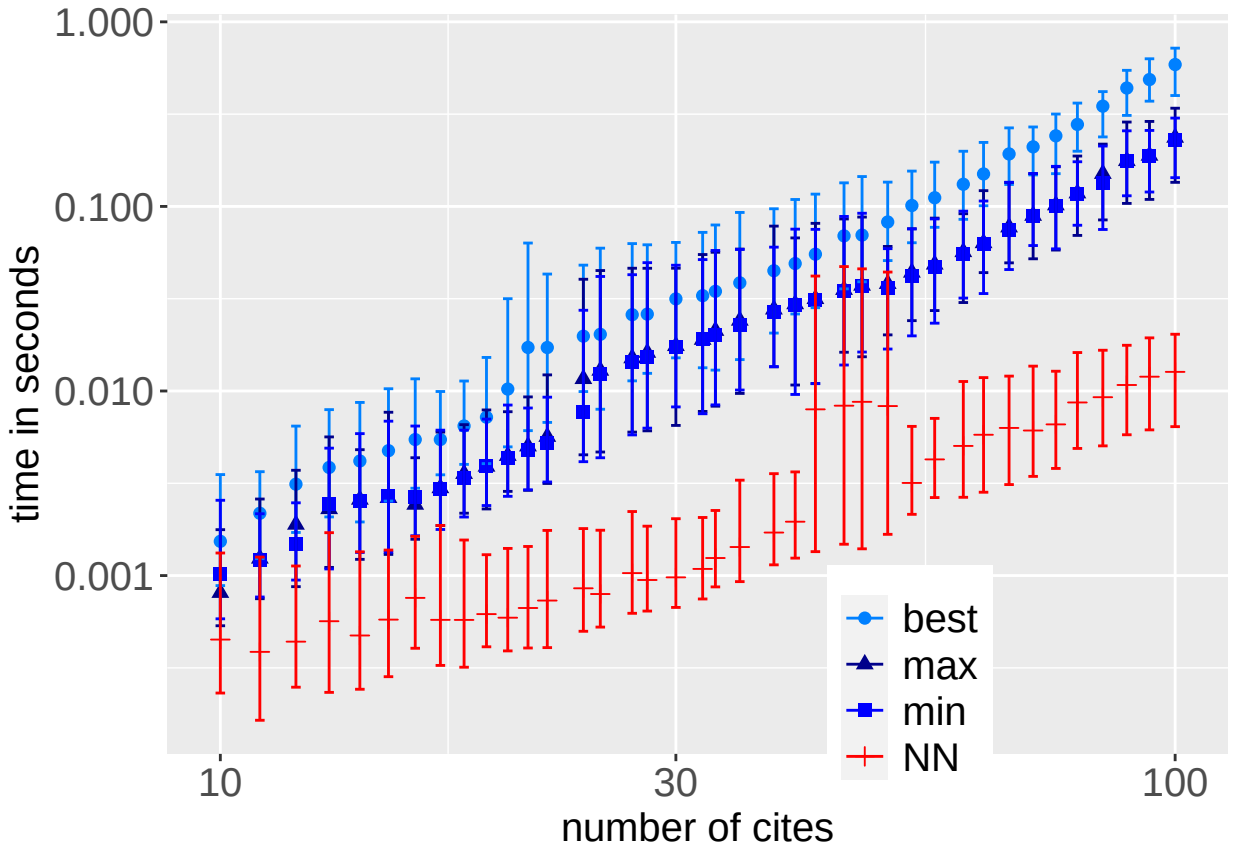
```
df_new <- data_summary(df2, varname="value",
                           groupnames=c("type","n"))
```

```
## 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
## -0.50838 -0.28623 -0.07654  0.10451  1.02353
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -12.16390    0.35264  -34.49   <2e-16 ***
## log(n)        1.68727    0.09888   17.06   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4063 on 37 degrees of freedom
## Multiple R-squared:  0.8873, Adjusted R-squared:  0.8842
## F-statistic: 291.2 on 1 and 37 DF,  p-value: < 2.2e-16
```

```
l1$coefficients
```

```
## (Intercept)      log(n)
##  -12.163902    1.687271
```

Pour best :

```
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
## -0.21754 -0.10616 -0.03652  0.09429  0.41542
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -11.90923    0.12634  -94.26   <2e-16 ***
## log(n)        2.44105    0.03543   68.91   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1456 on 37 degrees of freedom
## Multiple R-squared:  0.9923, Adjusted R-squared:  0.9921
## F-statistic:  4748 on 1 and 37 DF,  p-value: < 2.2e-16
```

```
l2$coefficients
```

```
## (Intercept)      log(n)
##  -11.909229    2.441047
```

Pour min :

```
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
## -0.22180 -0.16198 -0.03538  0.11417  0.34868
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -12.13803    0.14833  -81.83   <2e-16 ***
## log(n)        2.29782    0.04159   55.25   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1709 on 37 degrees of freedom
## Multiple R-squared:  0.988,  Adjusted R-squared:  0.9877
## F-statistic:  3052 on 1 and 37 DF,  p-value: < 2.2e-16
```

```
l3$coefficients
```

```
## (Intercept)      log(n)
##  -12.138032    2.297816
```

Pour max :

```
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
## -0.29502 -0.16687 -0.03215  0.13975  0.36329
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -12.13695    0.16457  -73.75   <2e-16 ***
## log(n)        2.30630    0.04614   49.98   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1896 on 37 degrees of freedom
## Multiple R-squared:  0.9854, Adjusted R-squared:  0.985
## F-statistic:  2498 on 1 and 37 DF,  p-value: < 2.2e-16
```

```
l4$coefficients
```

```
## (Intercept)      log(n)
##  -12.136954    2.306298
```

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