TP M2 Data Science
Le problème du voyageur de commerce
R versus Rcpp

Mohamed NIANG

26 septembre 2019

Le problème du voyageur de commerce

Présentation

On considère n villes réparties aléatoirement dans le plan. Notre objectif est de trouver le chemin le plus court partant de la première ville, visitant toutes les villes et revenant à la première.

Nous avons écrit un package nommé `salesmanRcpp` qui donne la solution exacte de ce problème en utilisant des fonctions écrites en Rcpp.

Le package peut être téléchargé par la commande suivante (enlever le #)

```
#devtools::install_github("Niangmohamed/salesmanRcpp")
```

puis installé ainsi

```
library(salesmanRcpp)
```

La création d'un jeu de n villes se fait avec la fonction '`Towns` suivante:

```
myTowns <- Towns(6)
myTowns
```

```
##             X          Y
## 1  2.97651827 -0.9810026
## 2  0.18893533  0.8373328
## 3 -3.00326474  0.2680984
## 4  0.49839492 -0.7398072
## 5 -0.62307913  0.4520653
## 6  0.02051234 -0.3456688
```

La position des villes est simplement un tirage aléatoire gaussien centré réduit pour X et pour Y. Tous les tirages sont indépendants.

Problématique et solution naïve

La difficulté du problème du voyageur de commerce réside dans sa très grande complexité algorithmique de l'ordre de $O(n!)$. Complexité dite factorielle. On trouvera ici quelques précisions sur la notion de complexité des algorithmes.

Une première idée naïve laisserait penser qu'il suffit de considérer, à chaque pas, la ville la plus proche. Par quelques dessins on peut se convaincre que cette solution n'est pas optimale.

La bonne solution naïve consiste à considérer toute les permutations p de l'ensemble $\{2, ..., n\}$ et de calculer à chaque fois la distance du chemin $(1, p, 1)$.

On a implémenté ces deux solultions dans les fonctions `monVoyageClosest` et `monVoyageR`.

On considére le problème avec seulement 4 villes et on calcule 100 fois les chemins “optimaux” avec les 2 algorithmes. On s’aperçoit qu’un nombre important de configurations donnent des résultats différents. `monVoyageClosest` échoue souvent à découvrir le chemin le plus court.

```
test <- NULL
for(i in 1:100)
{
  myTowns <- Towns(4)
  VR <- monVoyageR(towns = myTowns)
  VC <- monVoyageClosest(towns = myTowns)
  test <- c(test, all(VR == VC) || all(VR == rev(VC)))
}
```

Le nombre de résultats égaux entre les 2 algorithmes est égal à

```
sum(test)/100
```

```
## [1] 0.61
```

On illustre l’échec de `monVoyageClosest` sur un exemple avec en bleu le résultat obtenu avec `monVoyageR` et en rouge le résultat obtenu avec `monVoyageClosest`.

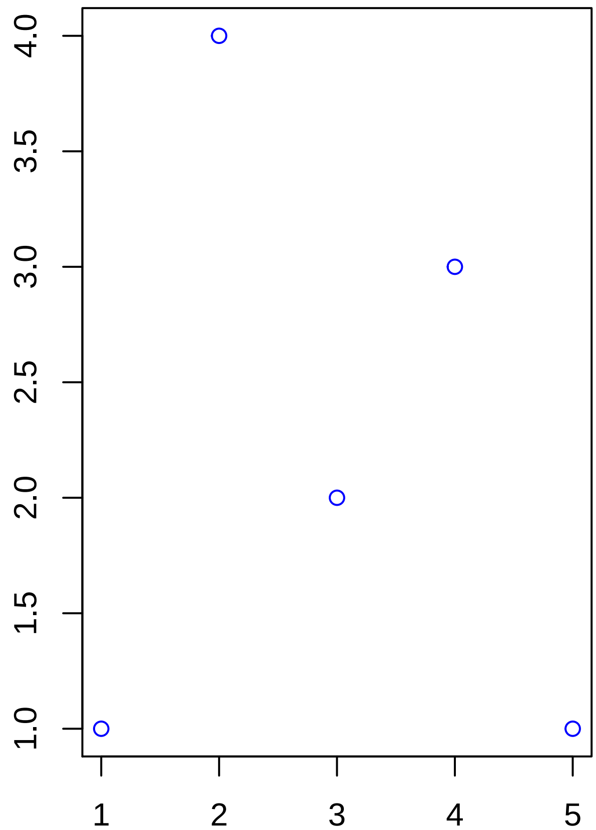
```
myTowns <- Towns(4)
VR <- monVoyageR(towns = myTowns)
VC <- monVoyageClosest(towns = myTowns)
test <- all(VR == VC) || all(VR == rev(VC))

while(test == TRUE)
{
  myTowns <- Towns(4)
  VR <- monVoyageR(towns = myTowns)
  VC <- monVoyageClosest(towns = myTowns)
  test <- all(VR == VC) || all(VR == rev(VC))
}
par(mfrow=c(1,2), mar=c(2,2,1,1))
plot(x = VR, towns = myTowns, col = "blue")
```

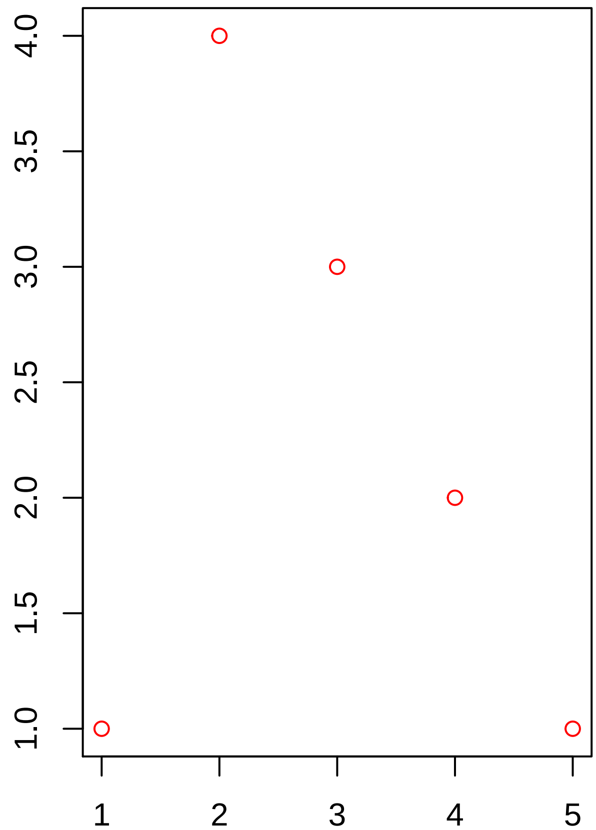
```
## Warning in plot.window(...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in plot.xy(xy, type, ...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "towns" n'est
## pas un paramÃ¨tre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "towns" n'est
## pas un paramÃ¨tre graphique

## Warning in box(...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in title(...): "towns" n'est pas un paramÃ¨tre graphique
```

```
plot(x = VC, towns = myTowns, col = "red")
```

```
## Warning in plot.window(...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in plot.xy(xy, type, ...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "towns" n'est
## pas un paramÃ¨tre graphique

## Warning in axis(side = side, at = at, labels = labels, ...): "towns" n'est
```

```
## pas un paramÃ¨tre graphique

## Warning in box(...): "towns" n'est pas un paramÃ¨tre graphique

## Warning in title(...): "towns" n'est pas un paramÃ¨tre graphique
```





On verra plus loin que la complexité algorithme factorielle pour être rédute à une complexité algorithmique exponentielle par l'algorithme de programmation dynamique de Held-Karp.

La description de ce dernier algorithme peut se trouver ici avec de nombreux autres compléments sur le problème du voyageur de commerce.

R versus Rcpp version naïve

Un premier test

Notre objectif étant de pouvoir calculer la solution pour des problèmes de complexité croissance ($n > 4$) il est naturel de rechercher à optimiser le code en utilisant C++. On compare ci-dessous le temps de calcul pour 7 villes répétés 10 fois entre la nouvelle fonction Rcpp `monVoyageRcpp` et la fonction en R `monVoyageR`.

```
timeR <- 0
timeRcpp <- 0
for(i in 1:10)
{
  myTowns <- Towns(7)
  timeR <- timeR + timeSalesmanR(myTowns)
  timeRcpp <- timeRcpp + timeSalesmanRcpp(myTowns)
}
```

Temps moyen pour 10 villes avec `monVoyageR`

```
timeR/10
```

```
## [1] 3.785595
```

Temps moyen pour 10 villes avec `monVoyageRcpp`

```
timeRcpp/10
```

```
## [1] 0
```

Le ratio moyen du temps de calcul de `monVoyageRcpp` sur`monVoyageR` pour 10 simulations est de

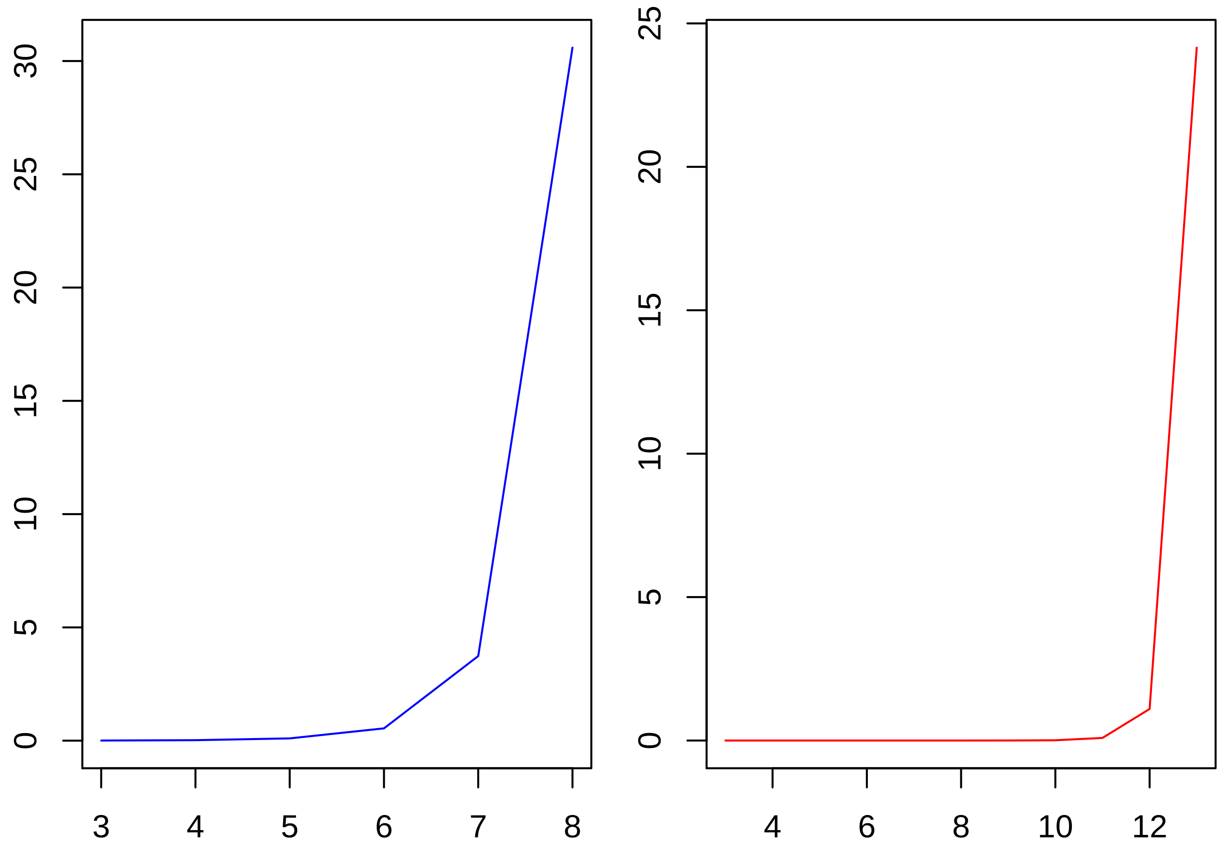
```
timeR/timeRcpp
```

```
## [1] Inf
```

n augmente

On trace en fonction de n le temps de résolution d'un problème avec `monVoyageR` puis `monVoyageRcpp`.

```
timeR <- NULL
timeRcpp <- NULL
exploreR <- 3:8
for(i in exploreR)
{
  myTowns <- Towns(i)
  timeR <- c(timeR, timeSalesmanR(myTowns))
}
exploreRcpp <- 3:13
for(i in exploreRcpp)
{
  myTowns <- Towns(i)
  timeRcpp <- c(timeRcpp, timeSalesmanRcpp(myTowns))
}
par(mfrow=c(1,2), mar=c(2,2,1,1))
plot(exploreR, timeR, type = 'l', col = "blue")
plot(exploreRcpp, timeRcpp, type = 'l', col = "red")
```



8 est une limite qu'il est difficile de dépasser avec mon ordinateur pour `monVoyageR`. On obtient un temps équivalent (20 s) pour 13 villes avec `monVoyageRcpp`.

L'algorithme de programmation dynamique de Held-Karp

Nous avons observé que le nombre de villes qu'un ordinateur portable standard peut considérer en un temps raisonnable (quelques minutes) est assez limité. Un algorithme (désormais non naif) qui permet une résolution exacte plus efficace car en temps exponentiel et non pas factoriel s'appelle l'algorithme de Held-Harp. Il se base sur le principe de la programmation dynamique, à savoir, il utilise le fait que les sous-problèmes (aller d'une ville i à j) sont résolus de manière optimale par l'algorithme. Dans le cas contraire on remplacerait le chemin trouvé par `monVoyageRcpp` par un autre chemin plus court.