Simulated Annealing applied to Travelling Salesmen

Mingyu Gai G27286107

1. Introduction

The traveling salesman problem presents the task of finding the most efficient and shortest route through a set of given cities. Each city should be passed through only once, and the salesman should go back to the starting city. In this research, we will use simulated annealing method to find the optimal route for the salesman. Firstly, choose a random route for the salesman and slightly modify the route by switching the order of two cities to obtain a new tour. If this new route's distance is shorter than the previous route, it becomes the new route for the salesman. Else, if this new route has a longer distance than the previous one, there exists some probability that this new tour still can be accepted anyway. When the difference between the previous route and new route has decreased and nearly equal to 0, the final result has come out.

2. Method and Algorithm

(1) The initial definitions and parameters:

Previous Location(c): the random route (initial route) for the salesman that through all the cities

New Location (c'): the route obtained by randomly switching two cities

Distance Function (h(c)): compute the total length of a location (Euclidian distance)

Relative Change in Distance (σ): the relative change in distance between c and c'

Temperature (t_0): temperature that use in the function at the first time

Temperature (t_k): temperature at the k^{th} instance of accepting a new solution state

Cooling Rate (α): the rate that the temperature lowered each time, $\alpha \in (0,1)$

Probability Function (p): determines the probability of moving to the new location

(2) Pseudocode:

1) Choose a random state c and define the initial t_0 and α

2) Create a new state c' by randomly swapping two non-adjacent edges in c

3) Compute $\sigma = h(c') - h(c)$

If $\sigma \leq 0$, then c = c'

If $\sigma > 0$, then assign c=c' with probability $P(\sigma, t) = e^{\frac{-\sigma}{t_k}}$

Compute $t_{k+1} = \alpha * t_k$ and increment the temperature iterations

4) Repeat steps 2 and 3 keeping track of the best solution until stopping conditions are met

(3) Stopping condition

The stopping conditions are quite important in simulated annealing. If the algorithm is stopped too soon, the approximation won't be as close to the global optimum, and if it isn't stopped soon enough, wasted time and calculations are spent with little to no benefit.

3.Process and Code in R

In my function, firstly I take the initial state randomly. In the following code, the location1 and location2 are the first permutation of the two problems. I use *"sample"* method to get the initial state randomly. The *"location"* matrix has 2 rows and 100 columns, each column represents the specific coordinate for each city. The column names stand for the name for each city and also could tell us the order of the cities.

```
> x <- rep(1:10, each = 10)
> y <-  rep(1:10, 10)
> location1 <- as.matrix(rbind(x, y))
> location2 <- as.matrix(rbind(X,Y))
> rownames(location1) <- NULL
> rownames(location2) <- NULL
> colnames(location1) <- sample(c(1:100), size = 100, replace = FALSE)
> colnames(location2) <- sample(c(1:100), size = 100, replace = FALSE)
```

After selecting the first state, we compute the distance of each pair cities and get 100*100 distance matrix. The aim of doing this could be simple for us to get the distance of each pair cities, instead of computing the distance each time for later iterations. The distances of each pair of cities are stored in the "*distance.matrix".*

```
> city.distance <- function(location){
  distance.matrix <- matrix(0, ncol = 100, nrow = 100)
  for (i in seq(1:ncol(location))){
    for (j in seq(1:ncol(location))){
      distance.matrix[i, j] = sqrt(sum((location[, i]-location[, j])^2))
    }
  }
  return(distance.matrix)
}
> distance.matrix1 <- city.distance(location1)
> distance.matrix2 <- city.distance(location2)
```

Next, compute the path distance of one specific state. The salesman will go through all the cities and return back to the starting point. The function we could see in the following. The *"cities"* stands for the order of the cities and could match with the *"location"* matrix column names. The other parameter *"matrix"* represents the distance matrix of each pair of cities.

```
> distance <- function(cities, matrix){
  d <- 0
  for (i in seq(1:(length(cities)-1))){
    d = d + matrix[cities[i], cities[i+1]]
  }
  d = d + matrix[cities[length(cities)], cities[1]]
  return(d)
}
```

Then compute the different path distance. In order to optimize the algorithm, we haven't need to compute all the distance of the path, we just need to compute the difference between the previous distance and the new distance. In my function, I randomly change two edges of the previous route and get the new route. For example, if the first route is 2, 3, 4, 5, 6, 7, 8, and we change city 3 and city 7, the new route will become 2, 7, 6, 5, 4, 3, 8. Hence, the edge 2→ 3 changes to 2→ 7, and 7→ 8 changes to 3→ 8. The rest edges do not change at all. So the difference of these two routes will be:

$$difference = \big(edge(2,3) + edge(7,8)\big) - (edge(2,7) + edge(3,8))$$

In addition, there are some particular situations we need to consider. Such as, if we just change the 100th city and 1^{th} city, the path distance will not change since we just change the order of the cities but the relative location of each city will be constant. The code is in the following. The *"city.1"* and *"city.2"* represents the cities changed. This function will return to the new state and the difference between two states.

```
> diff.distance <- function(city.1, city.2, cities, matrix){
   new.cities <- cities
   if (city.1 == 1){
     city.1.pre <- 100
   }else{
     city.1.pre <- city.1 - 1
   }
   if (city.2 == 100){
     city.2.post <- 1
   }else{
     city.2.post <- city.2 + 1
   }
   if((city.2 - city.1) == 99){
     diff <- 0
   }else{
     prev.distance <-  matrix[cities[city.1.pre],cities[city.1]] +
                       matrix[cities[city.2], cities[city.2.post]]
     new.distance <- matrix[cities[city.1.pre], cities[city.2]] +
                     matrix[cities[city.1], cities[city.2.post]]
     diff <- (new.distance - prev.distance)
   }
   for (swap in c(city.1 : city.2)) {
     new.cities[swap] <- cities[city.2 + city.1 - swap]
   }
    return(list(new.cities, diff))
 }
```

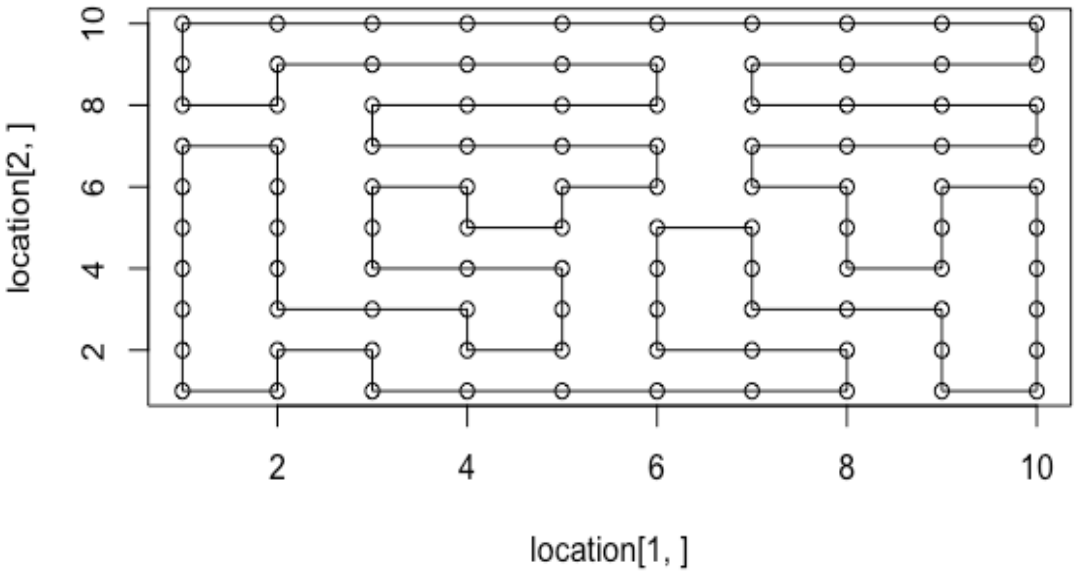
Finally, use simulated annealing method to get the optimal route for the salesman. We choose the initial temperature is 1 and the cooling rate is 0.99. Since when the iterations become larger and larger, the temperature will close to 0. Also, the rounding error will become bigger, then I set the function that the temperature change once when iterate 1000 times. If the probability of runif(1) is smaller than

```
> simulatedannealing <- function(cities, initial.temperature,
                                 cooling.rate, threshold, matrix){
   temperature <- initial.temperature
   iterations <- 1
   temperature.iterations <- 0
   last.distance <- distance(cities, matrix)
   while (iterations <= threshold){
     ## random swap two edges
     swap <- sample(cities, size = 2, replace = FALSE)
     city.1 <- min(swap)
     city.2 <- max(swap)
     ## compute the difference of the two cities
     diff <- diff.distance(city.1, city.2, cities, matrix)
     if(temperature.iterations >= 1000){
      ## each 1000 iterations change the temperature once
       temperature <- cooling.rate*temperature
       temperature.iterations <- 0
     }
     if (runif(1) < exp(-diff[[2]]/temperature)){
       cities <- diff[[1]]
       new.distance <- distance(cities, matrix)
      if (round(new.distance, 2) != round(last.distance, 2)) {
        last.distance <- new.distance
        }
     }
     iterations <- iterations + 1
     temperature.iterations <- temperature.iterations + 1
   }
   return(list(cities,new.distance))
 }
> cities <- seq(1, 100, by = 1)
> city1<-simulatedannealing(cities,1,0.99,1000000000, distance.matrix1)
>  city2<-simulatedannealing(cities,1,0.999,1000000000,  distance.matrix2)
```

4.Result and Plot

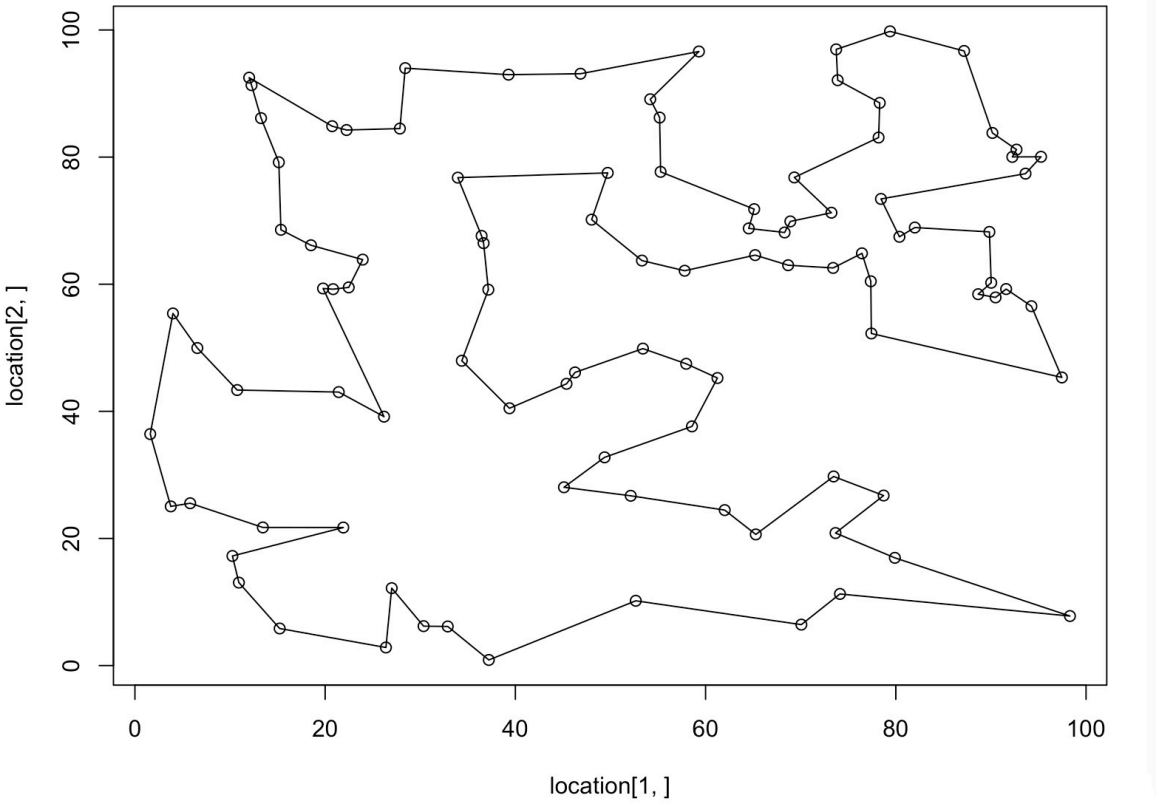
```
> new.location11 <- location1[, city1[[1]]]
> new.location22 <- location2[, city2[[1]]]
> plotcities <- function(location){
   plot(location[1, ],location[2, ])
   lines(location[1, ],location[2, ])
   lines(location[1, c(1,100)],location[2,c(1,100) ])
 }
> plotcities(new.location11)
> plotcities(new.location22)
```

After run the simulated annealing for large numbers of run, we get the final results. In first problem, we get the optimal route is 100. The plot and also the permutation are in the following. The permutation is also in the file1.




```
> new.location11
     87 74 69 36 34 4 17 95 50 100 58 28 96 93 37 78 39 67 76 80 33 46 64 92 98 16
[1,]  4  5  5  5  4 3  3  3  4   4  5  5  6  6  5  4  3  3  4  5  6  6  5  4  3  2
[2,]  2  2  3  4  4 4  5  6  6   5  5  6  6  7  7  7  7  8  8  8  8  9  9  9  9  9
     23 22 99 47 19 77 65 40 48 18 15 43 89 51 45 42 59 90 70 5 68 27 6 10 11 84 38
[1,]  2  1  1  1  2  3  4  5  6  7  8  9 10 10  9  8  7  7  8 9 10 10 9  8  7  7  8
[2,]  8  8  9 10 10 10 10 10 10 10 10 10 10 10  9  9  9  9  8  8 8  8  7 7  7  7  6  6
     3 44 91 30 14 31  7 55 85 20 66 56 57 29 54 52 94 9 8 79 35 82 2 53 75 41 71 73
[1,] 8  8  9  9  9 10 10 10 10 10 10 10  9  9  9  8  7  7 7 6  6  6  6 7  8  8  7  6  5
[2,] 5  4  4  5  6  6  5  4  3  2  1  1  2  3  3  3  4 5 5  4  3  2 2  2  2  1  1  1  1
     32 86 24 49 63 12 62 60 61 83 97 1 88 72 26 81 25 13 21
[1,]  4  3  3  2  2  1  1  1  1  1  1 1  2  2  2  2  2  3  4
[2,]  1  1  2  2  1  1  2  3  4  5  6 7  7  6  5  4  3  3  3
```

In the first problem, we get the optimal route is 767.6583. The plot and the permutation are in the following. The permutation is in the file2.




```
> location22
       30    12    16   100     1     3    67     5    34    47    35    27    77    62
[1,] 34.37 39.38 45.38 46.27 53.41 57.96 61.25 58.57 49.38 45.11 52.13 62.01 65.28 73.46
[2,] 47.96 40.48 44.33 46.12 49.87 47.48 45.24 37.63 32.76 28.05 26.71 24.46 20.63 29.75
       94    56    70    92     7    32     4    52    95    14    33    46    22    96
[1,] 78.72 73.66 79.90 98.31 74.13 70.07 52.66 37.21 32.89 30.35 27.00 26.39 15.22 10.93
[2,] 26.75 20.85 16.95  7.79 11.27  6.44 10.19  0.88  6.15  6.20 12.18  2.84  5.83 13.05
       73    75    78    43    84    28   93    31     8    13    68    23    97    42
[1,] 10.25 21.91 13.45  5.81  3.77  1.61  4.0  6.56 10.75 21.43 26.19 19.78 20.84 22.48
[2,] 17.24 21.72 21.73 25.54 25.04 36.42 55.4 49.96 43.35 43.03 39.16 59.33 59.22 59.50
       17    41    69    57    76    29     2    19    87    74    44    61    11    18
[1,] 23.95 18.51 15.35 15.13 13.26 12.26 12.01 20.74 22.25 27.86 28.42 39.28 46.85 59.30
[2,] 63.87 66.13 68.56 79.19 86.13 91.32 92.50 84.86 84.26 84.49 94.00 92.97 93.12 96.62
       48    86    26    89    60    51    80    54    98    49    36    15    72    25
[1,] 54.18 55.17 55.27 65.11 64.54 68.31 68.91 73.23 69.33 78.19 78.31 73.89 73.73 79.39
[2,] 89.10 86.22 77.68 71.81 68.78 68.14 69.88 71.25 76.80 83.08 88.53 92.10 96.95 99.79
       55    88    85    91     6    63    66    37    24    59    79    71    81    38
[1,] 87.18 90.16 92.69 92.25 95.27 93.64 78.44 80.38 82.00 89.84 90.03 88.66 90.49 91.60
[2,] 96.70 83.80 81.18 80.03 80.05 77.40 73.42 67.46 68.92 68.23 60.23 58.43 57.92 59.24
       45    83    20    65    53    50    82    58    40    39    10    99    90    21
[1,] 94.26 97.46 77.43 77.37 76.44 73.41 68.68 65.21 57.80 53.31 48.02 49.71 33.95 36.45
[2,] 56.54 45.33 52.25 60.46 64.86 62.56 62.99 64.58 62.12 63.72 70.16 77.52 76.77 67.60
        9    64
[1,] 36.66 37.16
[2,] 66.46 59.14
```

5. Discussion

In this project, we use simulated annealing method to get the optimal route for a travelling salesman. However, although this method could avoid the local maximum, it depends on the selecting of temperature, cooling rate, the initial state and also the number of iterations. If I had more time (or perhaps if I had a faster computer) I would run my function many more times for different values of cooling rate, temperature, and initial state to find the optimum combination for each problem. I run my function for lots of times in problem1, but only one time I get the best one 100. The probability of getting the optimal route I think is really small, full of fortune. In addition, the rounding error of the temperature after multiple the cooling rate for many times will increase. This is also a problem we need to consider.

Note: In my permutation file, since my initial state is selected randomly, so the order of the cities is different from you give us. Therefore, I will not only give you the permutation, but also the coordinate of the cities. Also, you can plot and compute the distance from my permutation file.