Solving the Travelling Salesman Problem with Simulated Annealing

Rachel Wood

2023-01-02

```
library(ggplot2)
library(tidyr)
library(gridExtra)
library(gganimate)

set.seed(-1)
```

Introduction

This portfolio will use the travelling salesman problem to illustrate the optimisation method of simmulated annealing, using the Tidyverse to store and display the necessary data. A quick overview of the problem, it's history and applications can be found in its Wikepedia page [1].

The Travelling Salesman Problem (TSP) Given n cities with a distance (or travelling time) given for each pair of cities (we assume these are symmetric), what is the shortest route which visits each city exactly once, ending at the starting city.

Creating and Visualising the Cities

We now use the Tidyverse to store city information and visualise them.

For determining the location of the cities we can use the following function, which selects n points within the unit square, using the `runif()` function

```
create_cities <- function(n){
  x <- runif(n)
  y<- runif(n)

  cities <- as_tibble(cbind(x, y))
  return(cities)
}
```

Then, given cities and a path stored as a vector of length n, we can create a function which will give us the length of the path

[1] "https://en.wikipedia.org/wiki/Travelling_salesman_problem"

```
distance <- function(cities, path){
  n<- length(path)
  order_cities <- cities[path,]

  order_cities1 <- cities[c(path[2:n], path[1]),]
  dist <- 0
  for (i in 1:n){
    dist <- dist + sqrt(sum((order_cities[i,] - order_cities1[i,])^2))
  }

  return(dist)
}
```

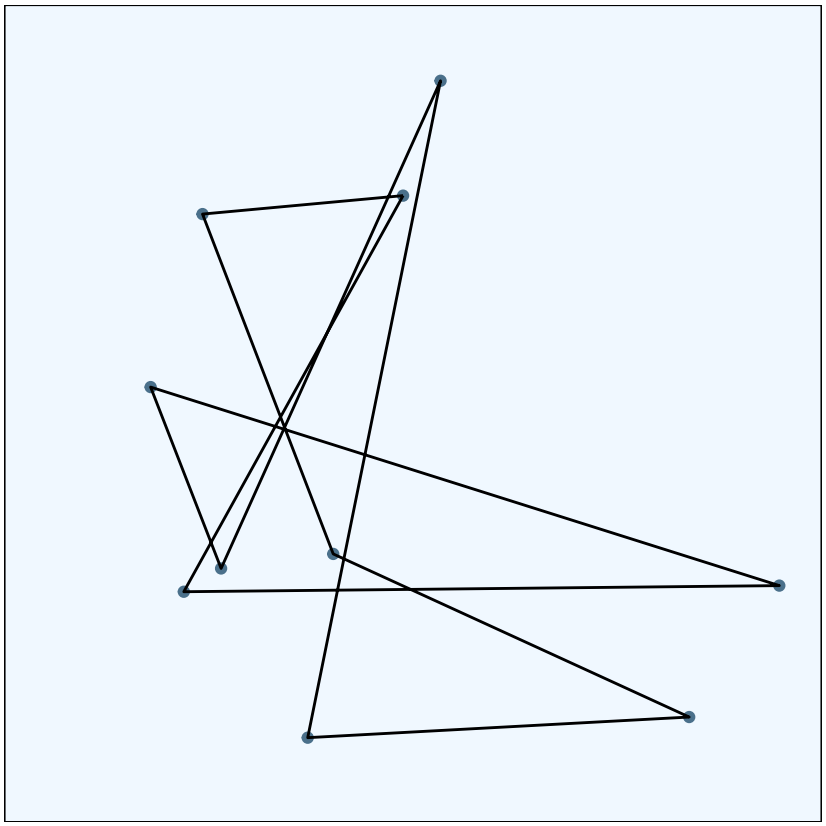
Now we can write a plotting function to view these

```
plot_cities <- function(cities){
  ggplot(cities, aes(x,y)) +
  geom_point(color = "skyblue4") +
  coord_fixed(ratio = 1, xlim= c(0,1), ylim = c(0,1)) +
  theme_void() +
  theme(panel.background = element_rect(fill = "aliceblue"))
}
```

We can then create a plot function to visualise a path.

```
plot_path <- function(cities, path, label = ""){
  path <- c(path, path[1])
  p <- plot_cities(cities)
  p + geom_path(data= as_tibble(cities[path,])) + ggtitle(label)

}
```

```
n <- 10
cities <- create_cities(n)

path <- as.vector(1:10)

plot_path(cities, path)
```



Simulated Annealing Method

Overview

The simulated annealing method is designed to find the global optimum (we assume minimum for simplicity). We start with a random state s, and choose a neighboring state s_{new} at random. We set $s \leftarrow s_{\text{new}}$ with probability $P(f(s), f(s_{\text{new}}), T)$ calculated from 3 factors:

- The value of the objective function at the original state $f(s)$
- The value of the objective function at the neighboring state $f(s_{\text{new}})$
- The global parameter T, called the temperature, which varies with time

A larger T will mean the probability of accepting a new state with higher objective value is higher, whereas a smaller T will make the probability of this happening smaller.

The algorithm starts with a large T, meaning we are more likely to 'jump around' the state space to avoid staying in a local optimum. The value of T then decreases with each iteration, approaching a steepest descent algorithm in the later iterations.

Choosing Parameters

There are a few things need to consider in the context of the TSP, the first being how to choose a neighboring state. For this we use the 2-opt algorithm, the steps for which are described below:

- A segment of the current path is selected (the start and end cities are chosen at random)
- Reverse the order of the cities in this segment

Next we consider how to decrease the temperature. In this example, we set it to decrease exponentially, by setting $T_{i+1} \leftarrow 0.9 \cdot T_i$, arbitrarily setting $T_0 = 100n$.

Finally we define the probability function $P(f(s), f(s_{\text{new}}), T)$. We want this to be such that for starting values of T will give a probability close to 1, and smaller values of T giving a probability close to 0. It also needs to increase with $f(s) - f(s_{\text{new}})$. For this we use the following:

$$P(f(s), f(s_{\text{new}}), T) = \begin{cases} 1 & f(s_{\text{new}}) < f(s) \\ \exp\left(\frac{f(s)-f(s_{\text{new}})}{T}\right) & \text{else} \end{cases}$$

Writing the Code

We now implement this all, first by writing the 2-opt function to find a neighbor

```
find_neighbour <- function(path){
  n <- length(path)

  start <- sample(1:n,1)
  if (start ==1) {
    end <- sample(2:n, 1)
  } else if (start ==n){
    end<- sample(1:(n-1),1)
  } else {
    end <- sample(c(1:(start-1), (start+1):n),1)
  }


  if (start < end){
    path[start:end] <- rev(path[start:end])
  } else {
    segment<- path[c(start:n, 1:end)]
    segment <- rev(segment)
    path[start:n] <- segment[1:(n-start+1)]
    path[1:end] <- segment[-(1:(n-start+1))]
  }
  return(path)

}
```

We also write a function to update the temperature

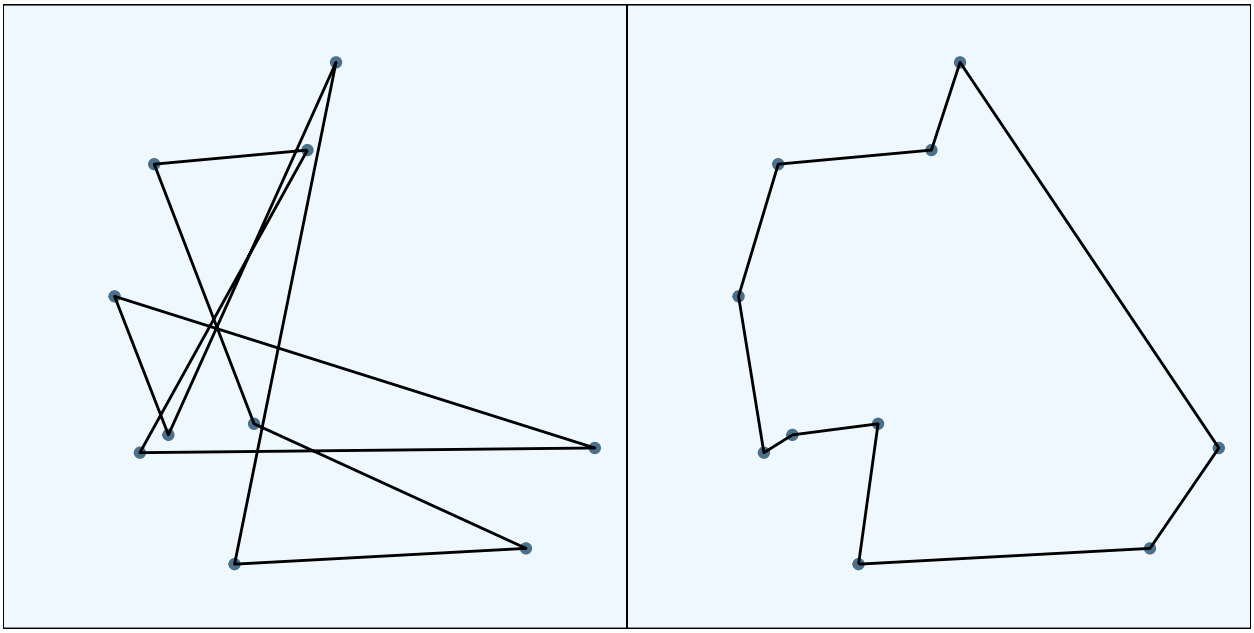
```
temp_change <- function(t){
  return(0.9*t)
}
```

Then the use this to write the simulated annealing function:

```
tsp_annealing <- function(start_path, cities, T0, maxIter){
  n <- nrow(cities)

  current_path <- start_path

  current_dist <- distance(cities, current_path)

  temp <- T0
  i <- 1

  while (i <= maxIter){
    new_path <- find_neighbour(current_path)
    new_dist <- distance(cities, new_path)

    rand <-runif(1)

    p <- exp(-(new_dist - current_dist)/temp)

    if (new_dist < current_dist | rand < p){
      current_path <- new_path
      current_dist <- new_dist
    }

    temp <- temp_change(temp)

    i <- i +1
  }


  return(list(path = current_path, distance = current_dist))
}
```

We can now run this function on our 10 cities, and compare the beginning and end paths:

```
start_plot <- plot_path(cities, path)
optimal <- tsp_annealing(path, cities, 100*length(path), 300)

end_plot <- plot_path(cities, optimal$path)

grid.arrange(start_plot, end_plot, nrow =1)
```



It might be interesting to visualise the way the path evolves through iterations, so instead of the final path, we can return a matrix with all the paths we have iterated through:

```
tsp_annealing_paths <- function(start_path, cities, T0, maxIter){
  n <- nrow(cities)

  paths <- matrix(nrow = maxIter, ncol = n)

  current_path <- start_path

  current_dist <- distance(cities, current_path)

  temp <- T0
  i <- 1

  p <- vector("list", floor(maxIter/100))
  while (i <= maxIter){
    new_path <- find_neighbour(current_path)
    new_dist <- distance(cities, new_path)

    rand <-runif(1)

    p <- exp(-(new_dist - current_dist)/temp)

    if (new_dist < current_dist | rand < p){
      current_path <- new_path
```

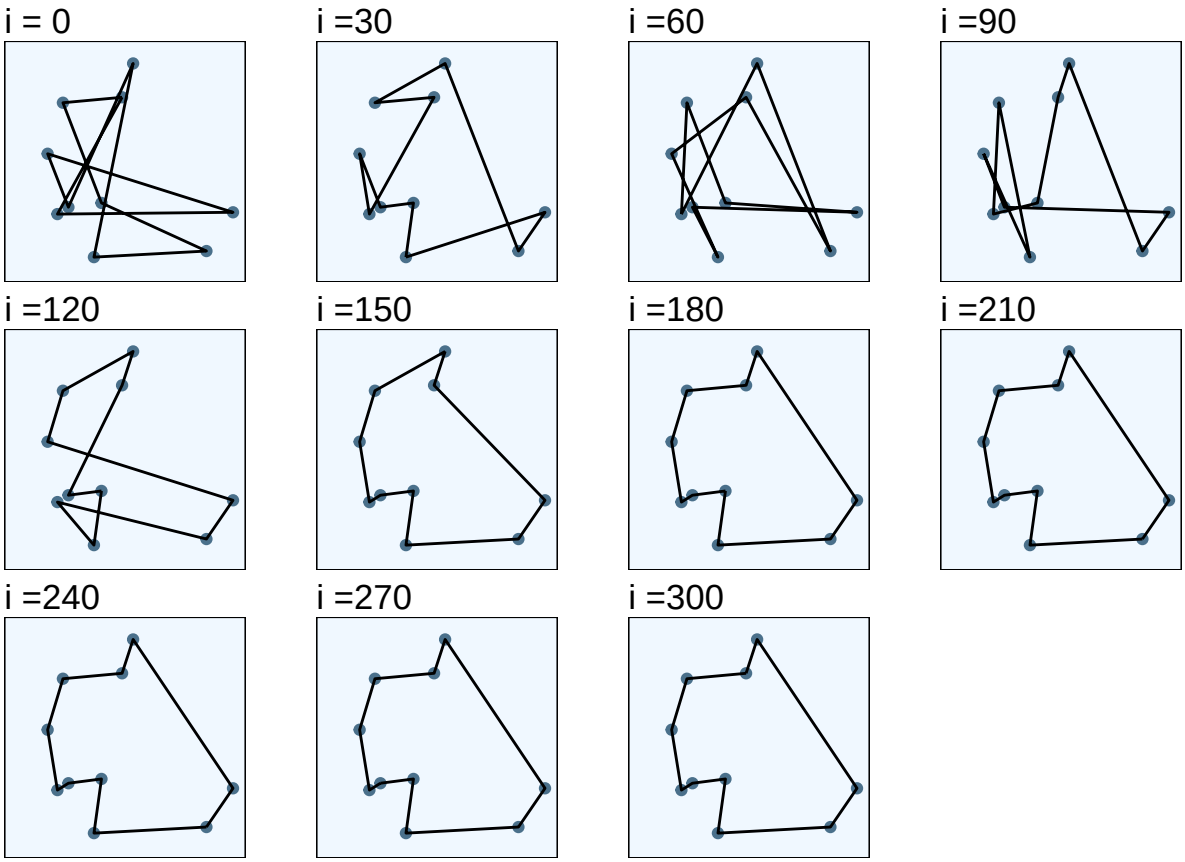
```
      current_dist <- new_dist
    }

    paths[i,] <- current_path
    temp <- temp_change(temp)

    i <- i +1

  }

  return(paths)

}
```

Results

We now view the results for $n = 10$ and $n = 30$ cities:

```
p <- tsp_annealing_paths(path, cities,100*length(path), 300)
```

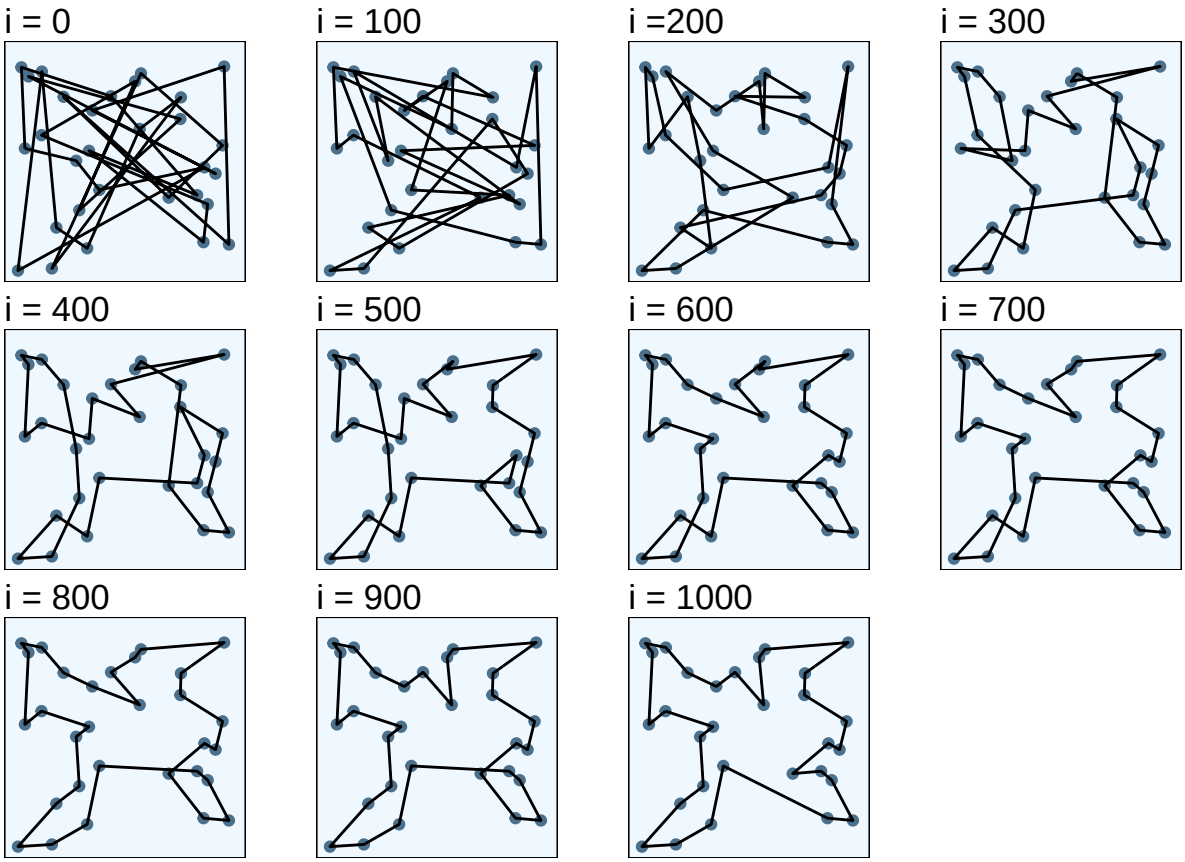
```
p_sub <- p[seq(30,300,30),]

grid.arrange(
  plot_path(cities, 1:10, label = "i = 0"),
  plot_path(cities, p_sub[1,], label = "i =30"),
  plot_path(cities, p_sub[2,], label = "i =60"),
  plot_path(cities, p_sub[3,], label = "i =90"),
  plot_path(cities, p_sub[4,], label = "i =120"),
  plot_path(cities, p_sub[5,], label = "i =150"),
  plot_path(cities, p_sub[6,], label = "i =180"),
  plot_path(cities, p_sub[7,], label = "i =210"),
  plot_path(cities, p_sub[8,], label = "i =240"),
  plot_path(cities, p_sub[9,], label = "i =270"),
  plot_path(cities, p_sub[10,], label = "i =300"),
  nrow=3

)
```



We now try the function on a larger set of cities:

```
n<- 30

cities<- create_cities(n)
path <- as.vector(1:n)

paths_30 <- tsp_annealing_paths(path, cities,100*length(path), 1000)
```

```
paths_sub_30 <- paths_30[seq(100,1000,100),]

grid.arrange(
  plot_path(cities, 1:30, "i = 0"),
  plot_path(cities, paths_sub_30[1,], "i = 100"),
  plot_path(cities, paths_sub_30[2,], "i =200"),
  plot_path(cities, paths_sub_30[3,], "i = 300"),
  plot_path(cities, paths_sub_30[4,], "i = 400"),
  plot_path(cities, paths_sub_30[5,], "i = 500"),
  plot_path(cities, paths_sub_30[6,], "i = 600"),
  plot_path(cities, paths_sub_30[7,], "i = 700"),
  plot_path(cities, paths_sub_30[8,], "i = 800"),
  plot_path(cities, paths_sub_30[9,], "i = 900"),
  plot_path(cities, paths_sub_30[10,], "i = 1000"),
  nrow=3
)
```



We can see it takes longer for the function to converge to the optimal solution, but from visual inspection, it seems like it has given the correct order of cities.