SIMULATION MODELLING FINAL PROJECT
A COMPARISON BETWEEN GENETIC ALGORITHM AND SIMULATED ANNEALING IN SOLVING TRAVELLING SALESMAN PROBLEM

GROUP 2 MEMBERS:

Aryamaan Jena	- BS18BDS009
Dam Ja Ba Nay	- BJ19BDS001
Jivitesh Sharma	- BS18BDS014
Amol Prakash	- BS18BDS004
Kansum Reddy	- BS18BDS003
Aditya Patil	- BS18BDS002
Hai Nguyen	- BJ19BDS003
Gautam Sadarangani	- BS18BDS015
Shivit Damija	- BS18BDS016

PROBLEM BACKGROUND

The travelling salesman problem is an optimization problem. It has a wide range of applications to fields like computer wiring, vehicle routing, combinatorial data analysis, etc.

The object of the travelling salesman problem is to find an optimum tour route wherein each of n specified cities (or places) must be travelled to, visiting each city exactly once and returning to the initial point. The optimum tour route is one in which the distance travelled in the route is minimized, where the distances between each city with the other n-1 cities is known.

Numerous solutions to this problem exist which focus on different aspects of combinatorial optimization. A few common simulation techniques used to solve the problem include genetic algorithms and simulated annealing. Each optimizes the tour route based on varying factors and parameters.

Simulated Annealing is a Monte-Carlo maximization or minimization technique used to find the overall minimum of a function that may possess several local minima. It mimics the process of annealing, which starts with a high temperature mix of metal and slowly cools the mix, allowing optimal structures to form as the material cool. It randomly generates a large number of possible solutions, keeping both good and bad solutions. As the simulation progresses, the requirements for replacing an existing solution or staying in the pool becomes stricter and stricter, mimicking the slow cooling of metallic annealing. Eventually, the process yields a small set of optimal solutions.

Genetic Algorithm adapts techniques from genetics in biology. Behaviours are encoded into genes and after each generation, models are tuned by allowing them to "breed" and "mate" with each other, as determined by a fitness value. In this process, genes are exchanged and crossovers or mutations can occur with some predefined probabilities. Then, the current population is discarded and its offspring forms the next generation.

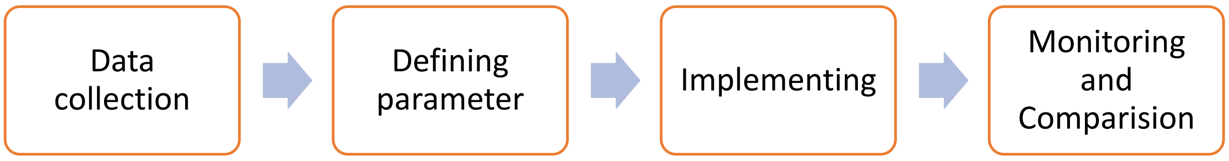
PROBLEM STATEMENT

In our project, we draw a comparative study between the performance of genetic algorithm and simulated annealing method in solving the travelling salesman problem. The factors considered when studying the two approaches includes the results of the algorithm, their performance as well as other points of comparison.

The genetic algorithm proves to give a better solution than simulated annealing. This however comes at the cost of time complexity as the performance time of genetic algorithms increases exponentially with the number of cities in the travelling salesman problem. Hence, while the genetic algorithm gives a better solution, it is much slower than simulated annealing, which in

turn gives a result with a lower runtime. We further discuss the methodology to arrive at these points herewith.

SOLUTION APPROACH



This project makes comparison of the two optimization algorithms, namely Genetic Algorithm and Simulated Annealing on the run time, the quality of solution and parameters used for each algorithm.

1. ***Dataset and data collection:***
   To make a valid comparison, we use the same distance matrix for both the algorithms. The data was generated using inbuilt R function "dist", which calculates the distance of each pair of cities for 10 cities in Australia given a list of city names.
2. ***Parameters:***
   Parameters are elements that can be tweaked or configured to explore the performance of the algorithm. They can be set as a constant throughout experiment or based on the

problem instance. In this project, to experiment how different values affect the final solutions and to get the optimized values for the travelling salesman problem, most of the parameters will be iterated within a predefined range. The parameters used for each algorithm in this project are described in the table below.

Table 1: Parameters used for Genetic Algorithm and Simulated Annealing

Genetic Algorithm	**Simulated Annealing**
Population	Temperature
Mutation rate	Cooling rate
Cut length	Absolute temperature

3. ***Monitoring runtime***
   While the main purpose is to come up with the optimized route for the salesman to travel, it is of utmost importance to take into consideration the runtime of each algorithm. Comparing runtime of two algorithms is a tough task due to the difficulties in the execution of the code in different compilers and machines with potential difference in configurations.
4. ***Implementation:***
   a. **Genetic algorithm:**
   From the above problem statement, we will be solving the travelling salesman problem using Genetic Algorithms. The way we implemented this code was by using the existing 'GA' library in RStudio. The GA function automates the whole process of Genetic Algorithms and gives an optimal route for the given input. Thus we start off by calling the library
   Then we initialized the **tourLength** function, which takes in 2 arguments, which is tour and distMatrix. The **tourLength** function gives the total distance covered from one city to another. It takes an input 'tour' which is a vector of different cities and gives the total distance between those cities.

   The **tspFitness** function is a fitness function to be used in the GA function as one of the parameters

   Then we call summary on GA object to see the results. In this last bit of code the GA function is called and fed with the different parameters to run it. For the distMatrix attribute, we use the **dist_mat** matrix we created earlier. And for the fitness attribute we apply the **tspFitness** function initialized above.

   b. **Simulated Annealing**

   The simulated annealing approach has 3 functions as explained below
   i. **distance():**
   This function takes in a sequence as input and return the total distance travelling in that sequence according to the distance matrix
   ii. **swap():**
   This function helps generate a new sequence each which doesn't exist before. All of this is done by randomization.
   iii. **acceptingProb():**
   This function shows the accepting probability concept in simulated annealing, it takes in 2 parameters, the difference between the current and

previous distance and the temperature and return the value for further decision. The formula is as follows: *1/(1+exp(delta_dist/temperature)*

Then the annealing process will be executed within a loop, starting off by generating new sequence. The distance for the new sequence will then be calculated and is fed to the **acceptingProb()** function to find the accepting probability. A random probability will then be generated to compare with the accepting probability, which helps get out of local minima. If the random probability less than or equal to the accepting probability, the model will then record the current results and move on to the next iteration.

CODE IMPLEMETATION

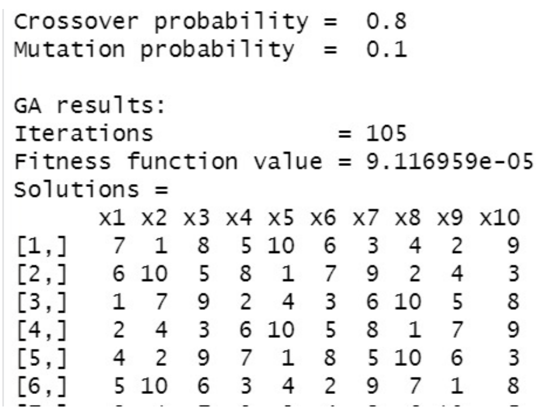
Please refer to the APPENDIX section for code implementation in R

RESULT AND DISCUSSION

1. Genetic Algorithm

After running summary (GA) this is the output we obtained:

From this output, we can see that function outputs the population size = 10, which is the 10 cities we selected. The number of generations shows the maximum possible iterations for this function. But as we can see, we got the solution in 149 iterations. The solution part shows the optimal path with the least distance starting from city 1.



```
Crossover probability =  0.8
Mutation probability  =  0.1

GA results:
Iterations             = 105
Fitness function value = 9.116959e-05
Solutions =
      x1 x2 x3 x4 x5 x6 x7 x8 x9 x10
[1,]   7  1  8  5 10  6  3  4  2   9
[2,]   6 10  5  8  1  7  9  2  4   3
[3,]   1  7  9  2  4  3  6 10  5   8
[4,]   2  4  3  6 10  5  8  1  7   9
[5,]   4  2  9  7  1  8  5 10  6   3
[6,]   5 10  6  3  4  2  9  7  1   8
```

To find the total distance we called the **tourLength()** function:

```
tourLength(GA@solution,dist_mat)
```

In this function the tour path is given in the form of a vector using ‘GA@solution’ and **dist_mat** is given as the second argument. The output is as follows:

```
[1] 10968.57
```

The given output above is the total distance given for the best solution.

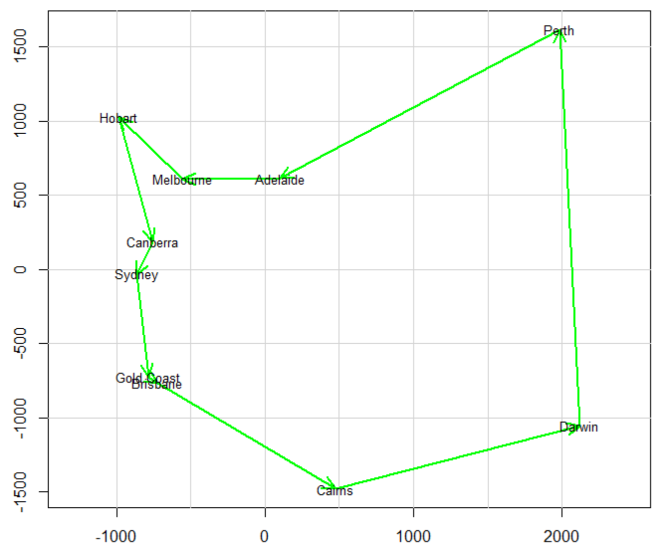
```
all_dist = c()
for (i in 1:length(GA@bestSol)){
     all_dist = c(all_dist,tourLength(GA@bestSol[[i]],dist_mat))
}
GA@bestSol[all_dist==min(all_dist)]
min(all_dist)
```

To see in how many iterations, we have the least distance covered, for that we use the above function. This function runs through every iteration and returns a vector with the total distance covered by each solution. By calling the **min()** function we find the minimum distance covered among them. The output from this is the same as **tourLength(GA@solution, dist_mat)**

Given below is the visual representation of the path to be followed:

To visualize the path we used the coordinates given for the 10 cities from the distance matrix
We also plotted the distance for best solution given in every iteration of the GA function

Figure 1: Paths through all 10 cities



From this graph it can be clearly seen that the distance keep decreasing with every couple of iterations but occasionally spikes up.

To understand how the GA function progresses over many simulations, we plotted it as shown below:

Figure 2: Distance for the best solutions

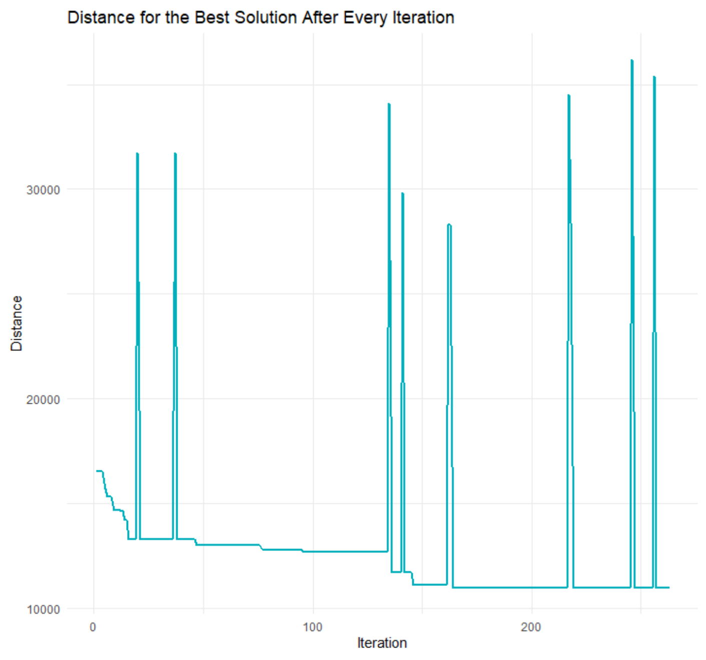


Figure 3: GA progression

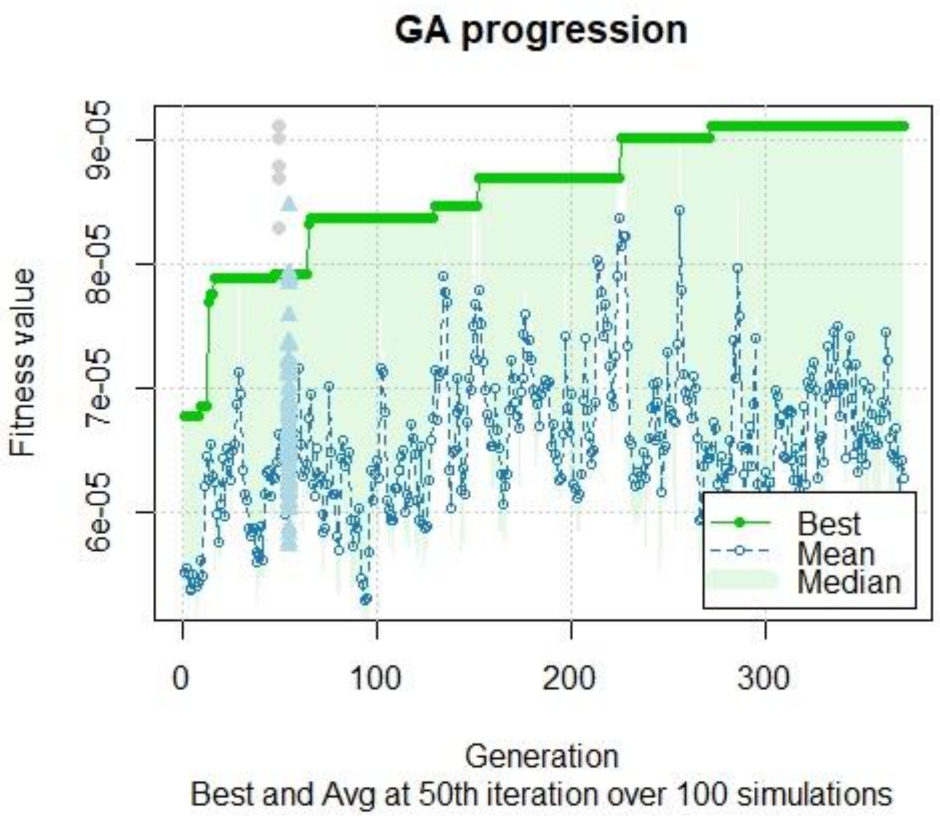


Figure 3 is the representation that compares the best and the average fitness values over 100 simulations. It can be seen that with every simulation, the fitness value keeps increasing. The green line shows the best fitness value which is on a steady rise as well as the mean fitness value (denoted as the blue line) which has a lot of fluctuations.

Runtime

To find the runtime for one simulation of the GA function we used the code below:

```
start_time= Sys.time()
GA <- ga(type = "permutation", fitness = tspFitness, distMatrix =
dist_mat, lower = 1, upper = attr(dist_mat,"dim"), popSize = 10,
maxiter = 500, run = 100, pmutation = 0.2, monitor = NULL, keepBest =
TRUE)
end_time = Sys.time()
runtime = end_time-start_time
runtime
```

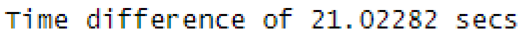
The output of this function:

```
Time difference of 0.2270229 secs
```

The output of this function may differ with every simulation and the number of iterations in each simulation

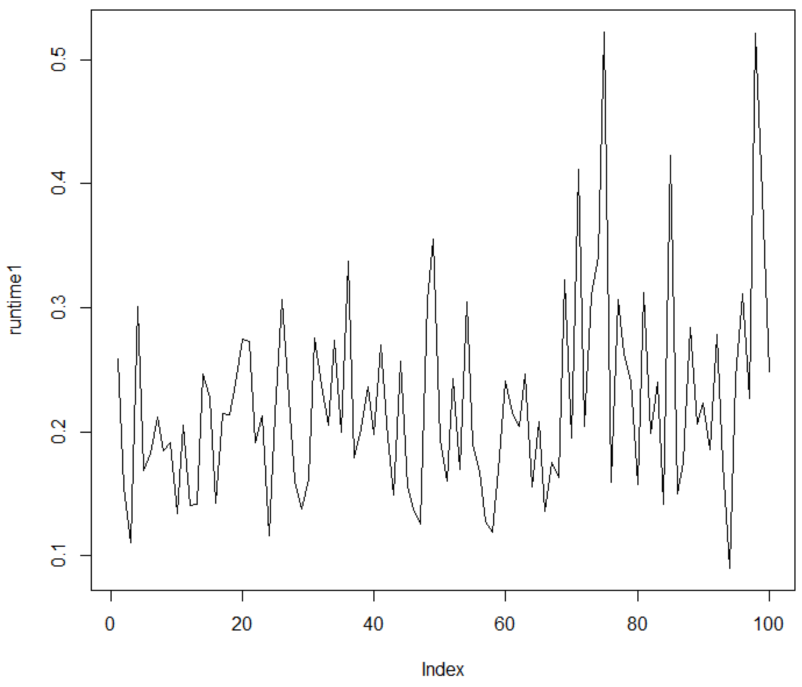
The same can be done to a simulation of the GA function that has 100 iterations:

```
start_time = Sys.time()
for (i in 1:100) {
	GA <- ga(type = "permutation", fitness = tspFitness, distMatrix
	dist_mat, lower = 1, upper = attr(dist_mat, "dim"), popSize =
	10, maxiter = 500, run = 100, pmutation = 0.2,monitor = NULL,
	keepBest = TRUE)
}
end_time = Sys.time()
runtime = end_time-start_time
runtime
```

The output for this is: 

```
Time difference of 21.02282 secs
```

Figure 4: Runtime difference

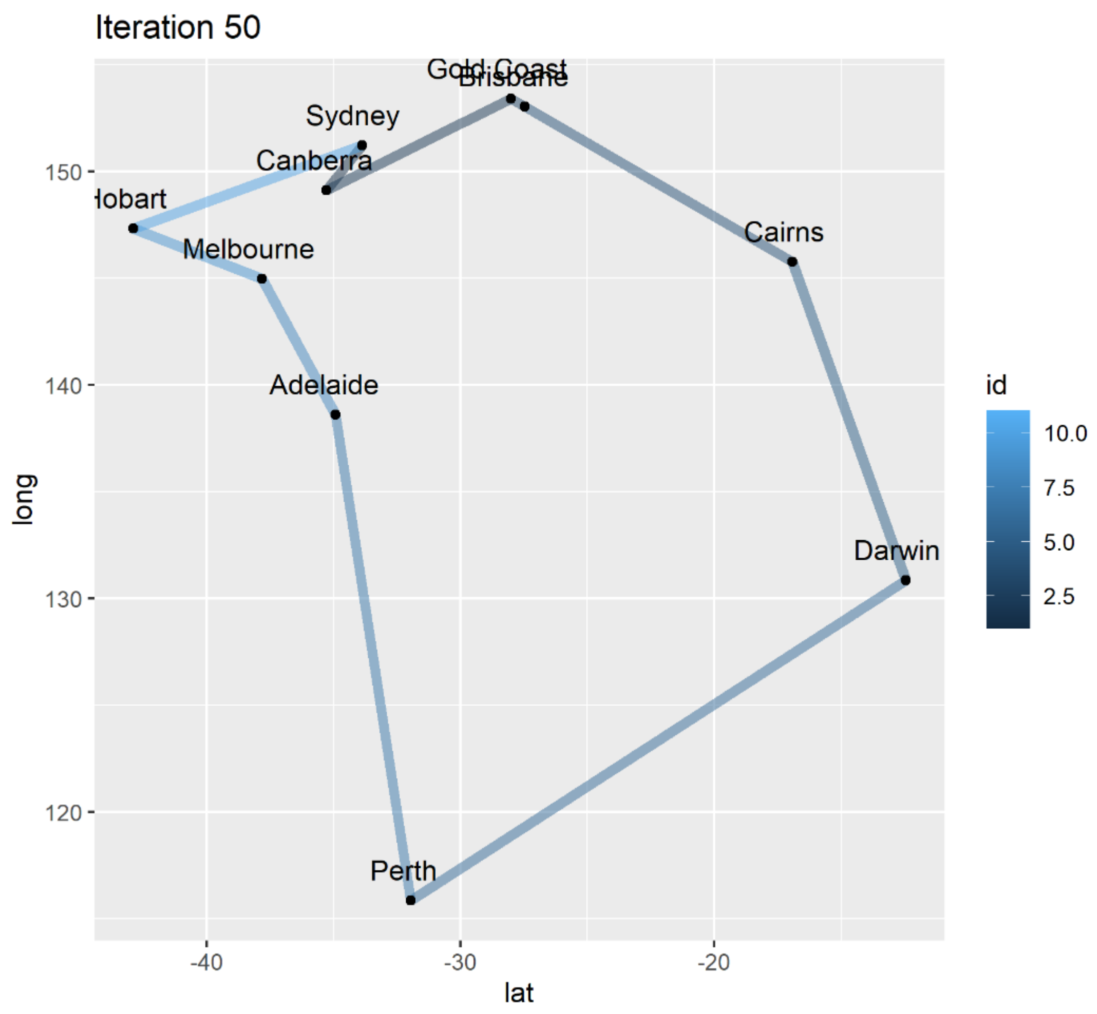


As you can see there is huge difference in run time when the simulations are increased to 100. The time output may differ depending on the number of iterations in each simulation.

As seen above from the plot, it visualizes the runtime of all the 100 simulations. For most simulations the runtime is around 0.1 seconds but peaks to 0.5 seconds a couple of times

Creating GA training progress animation.
By adding “keepBest = TRUE” inside the GA algorithm, we can keep the best result for each training iteration in the model.
We will create a DataFrame for the purpose of plotting and animating the plot.
And then due to the nature of geom_path, we will export and plots from a loop and then combining them into an animated Gif using Gifski or FFMpeg instead of using the GGanimate library.
The resulting Gif can be accessed here:
https://media.giphy.com/media/UPiBNaw3dQVIiToX96/giphy.gif

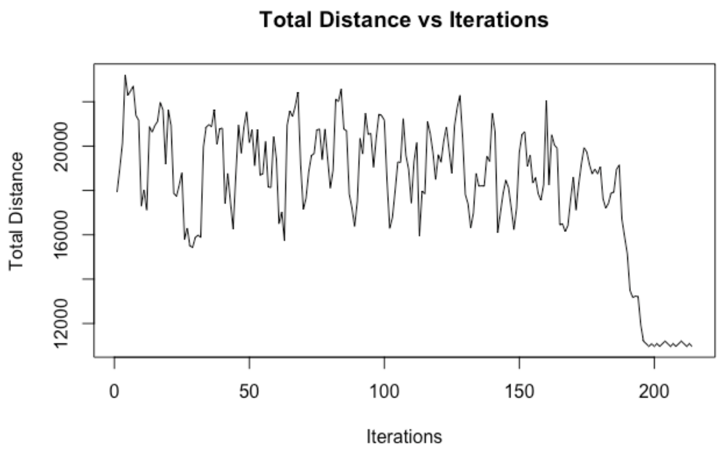
Figure 5: GIF for the best path



2. Simulated Annealing

After running our Simulated annealing, this is the results we get using the cooling rate of 10 percent.

Figure 6: Total distance versus Iterations



As we can see from this noisy graph, the error/ total distance bounces around for an extremely long time ~ 180 iterations before it converges around the minima

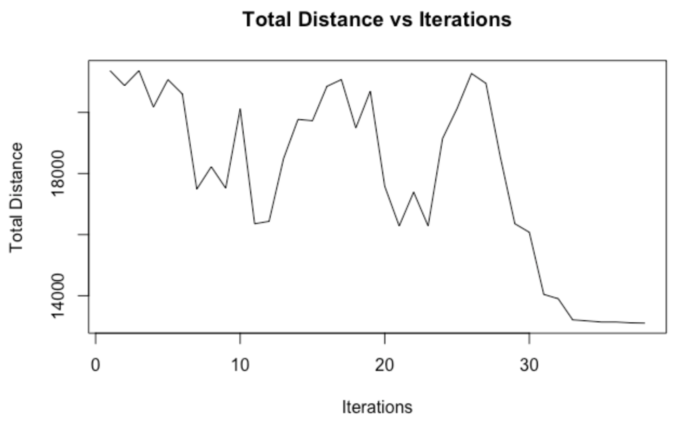
The solution for the travelling salesman problem provided by the algorithm is as follows

```
iteration:  215 Temperature:  16.13976
sequence:  1 7 2 9 4 3 6 10 5 8 1 , New sequence:  1 7 9 2 4 3 6 10 5 8 1
Total Distance:  11083.76 Delta Distance:  -115.19 , New Total Distance:  10968.57
```

Here we can see that the algorithm converges completely at the 215 iteration and find the minimum distance of 10968.57. This solution is the same one that we got from genetic algorithms the sequence

If we use a higher cooling rate of 50 percent, the algorithm may not converge to the local minima, and would be stuck at a local minima as shown below:

Figure 7: Total distance versus Iterations



Here we can see in this noisy graph, the algorithm does not run for more than 50 iterations. It also seems like the algorithm has found its minima at around 13000 which is much higher than the earlier 10968.57

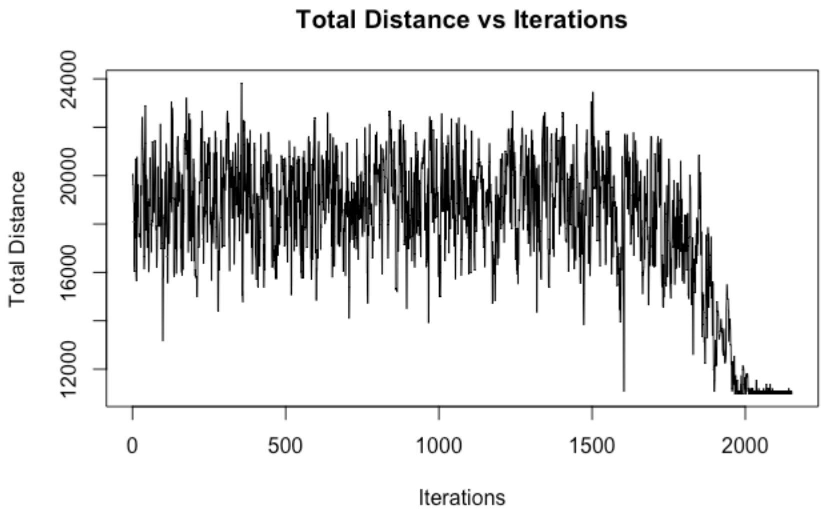
The results for this algorithm using cooling rate as 50 percent.

```
iteration:  37 Temperature:  1.455192
sequence:  7 4 3 6 10 2 9 1 5 8 7 , New sequence:  7 4 3 6 10 2 9 1 8 5 7
Total Distance:  13111.85 Delta Distance:  -8.49 , New Total Distance:  13103.36
```

Here we can see that the algorithm has stuck to the local minima of 13103.36. so we can conclude that if the cooling rate is high then the chances of our algorithm not converging to the minimum also become higher.

The idea of simulated annealing is to reduce the temperature slowly, so here we take the cooling rate of 1 percent.

Figure 8: Total distance versus Iterations



Here we see an extremely noisy graph that runs for over 2000 iterations but it does converge to the global minima.

Here we can see the output of the algorithm for when the cooling rate is 1 percent:

```
iteration:  2151 Temperature:  41.27322
sequence:  2 4 3 6 10 8 5 1 7 9 2 , New sequence:  2 4 3 6 10 5 8 1 7 9 2
Total Distance:  11086.34 Delta Distance:  -117.77 , New Total Distance:  10968.57
```

We can see that the number of iterations has greatly increased. It has also converged to the global minima of 10968.57. A lower cooling rate helps with not getting stuck in the local minima.

Table 2: Runtime of different cooling rate

Cooling Rate	**Time taken(s)**
0.01	2.054806
0.1	2.040282
0.5	2.04394

The table 2 shows us the runtime of code execution with different values for the colling rate, as we can see, the time taken is roughly the same.

Table 3: Parameters with best solution

Genetic Algorithm	**Simulated Annealing**
Best solution: 10968.57	Best solution: 10968.57
Iterations: 105	Iterations: 215
Mutation rate: 0.1	Cooling rate: 0.1
Crossover rate: 0.8	Temperature: 16.13976

CONCLUSIONS

In this project, the two algorithms, namely Genetic Algorithm and Simulated Annealing have been discussed with an aim to compare their performance on solving the Travelling Salesman problem. Experiments have been conducted to observe their performance on different testing configurations. In general, both algorithms are very good solvers and can provide optimal solutions if the right values of parameters are set. Both algorithms come up with the same answer in this experiment and can be used to solve practical optimization problem in real life like the warehousing distribution problem.

REFERENCE

Top cities in Australia: https://www.bookmundi.com/t/12-best-cities-to-visit-in-australia

APPENDIX

```
city_names<- c("Sydney","Melbourne","Perth","Adelaide","Brisbane","Darw
in","Canberra","Gold Coast" ,"Hobart","Cairns")
dist_mat <- matrix(0, nrow = 10,ncol = 10)
dimnames(dist_mat)<-list(city_names, city_names)

# sydney - and all cities
dist_mat[1,2]<- dist_mat[2,1]<-  712.97 # sydney - melnb
dist_mat[1,3]<-dist_mat[3,1]<-3290.40 # sydney - perth
dist_mat[1,4]<-dist_mat[4,1]<-1161.71 # sydney - adelide
dist_mat[1,5]<-dist_mat[5,1]<-732.82 # sydney - brisbane
dist_mat[1,6]<-dist_mat[6,1]<-3148.95 # sydney - darwin
dist_mat[1,7]<-dist_mat[7,1]<-246.09 # sydney - canberra
dist_mat[1,8]<-dist_mat[8,1]<-686.36 # sydney - gold coast
dist_mat[1,9]<-dist_mat[9,1]<-1057.04 # sydney - hobart
dist_mat[1,10]<-dist_mat[10,1]<-1961.29 # sydney - cairns

# melb and all cities
dist_mat[2,3] <- dist_mat[3,2] <-2721.74 #melb - perth
dist_mat[2,4] <-dist_mat[4,2] <- 653.81 #melb - adelaide
dist_mat[2,5] <-dist_mat[5,2] <- 1374.43 #melb -brisbane
dist_mat[2,6] <-dist_mat[6,2] <-3148.21 #melb - darwin
dist_mat[2,7] <-dist_mat[7,2] <-466.89 #melb - canberra
dist_mat[2,8]<-dist_mat[8,2] <-1345.93 #melb - gold coast
dist_mat[2,9] <- dist_mat[9,2] <-597.77 #melb - hobart
dist_mat[2,10] <- dist_mat[10,2] <-2324.49 #melb - cairns

# perth and all cities
dist_mat[3,4]<-dist_mat[4,3]<- 2131.23 # perth - adeliade
dist_mat[3,5]<- dist_mat[5,3]<-3606.17 # perth -brisbanr
dist_mat[3,6]<- dist_mat[6,3]<-2653.25 # perth - darwin
dist_mat[3,7]<- dist_mat[7,3]<-3088.79 # perth - canberra
dist_mat[3,8]<- dist_mat[8,3]<-3627.40 # perth - gold coast
dist_mat[3,9]<- dist_mat[9,3]<-3010.29 # perth -hobart
dist_mat[3,10]<- dist_mat[10,3]<-3441.04 # perth - cairns

# adelaide and all cities
dist_mat[4,5]<- dist_mat[5,4]<-1600.41 #adelaide - brisbane
dist_mat[4,6]<-dist_mat[6,4]<- 2617.46 #adelaide -darwin
```

```r
dist_mat[4,7]<- dist_mat[7,4]<-958.44 #adelaide -canberra
dist_mat[4,8]<- dist_mat[8,4]<-1600.95 #adelaide -gold coast
dist_mat[4,9]<- dist_mat[9,4]<-1161.53 #adelaide -hobart
dist_mat[4,10]<-dist_mat[10,4]<- 2124.82 #adelaide -cairns

# brisbane and all cities
dist_mat[5,6]<- dist_mat[6,5]<-2847.49 #brisbane - darwin
dist_mat[5,7]<-dist_mat[7,5]<- 943.58 #brisbane - canberra
dist_mat[5,8]<- dist_mat[8,5]<-71.31 #brisbane - gold coast
dist_mat[5,9]<- dist_mat[9,5]<-1789.15 #brisbane - hobart
dist_mat[5,10]<- dist_mat[10,5]<-1389.44 #brisbane - cairns

# darwin and all cities
dist_mat[6,7]<-dist_mat[7,6]<- 3133.49 #darwin - canberra
dist_mat[6,8]<- dist_mat[8,6]<-2912.47 #darwin  - gold coast
dist_mat[6,9]<- dist_mat[9,6]<-3735.97 #darwin  - hobart
dist_mat[6,10]<- dist_mat[10,6]<-1679.89 #darwin - cairns

# canberra and all cities
dist_mat[7,8]<- dist_mat[8,7]<-905.61 #canberra - gold coast
dist_mat[7,9]<-dist_mat[9,7]<-859.42 #canberra - hobart
dist_mat[7,10]<- dist_mat[10,7]<-2068.32 #canberra -cairns

# gold coast and all cities
dist_mat[8,9]<- dist_mat[9,8]<-1743.32 # gold coast - hobart
dist_mat[8,10]<- dist_mat[10,8]<-1460.75 # gold coast - cairns c

#hobart and all cities
dist_mat[9,10] <-dist_mat[10,9]<- 2890.27 # hobart - cairns

dist_mat
"-----------------------------------------------------------------------
-----------------------------"
#running the GA function
library(GA)

tourLength <- function(tour, distMatrix) {
  tour <- c(tour, tour[1])
  route <- embed(tour, 2)[, 2:1]
  sum(distMatrix[route])
}


tspFitness <- function(tour, ...) 1/tourLength(tour, ...)


# run a GA algorithm

start_time= Sys.time()
```

```r
GA <- ga(type = "permutation", fitness = tspFitness, distMatrix = dist_
mat, lower = 1,
             upper = attr(dist_mat,"dim"), popSize = 10, maxiter = 500,
 run = 100, pmutation = 0.2,
             monitor = NULL, keepBest = TRUE)
end_time = Sys.time()
runtime = end_time-start_time

runtime

summary(GA)
tourLength(GA@solution,dist_mat) #finding total distance
"-----------------------------------------------------------------------
--------------------------------"
all_dist = c()
start_time= Sys.time()
for (i in 1:100) {
  all_dist = c(all_dist,tourLength(GA@bestSol[[i]],dist_mat))
}

GA@bestSol[all_dist==min(all_dist)]

min(all_dist)
"-----------------------------------------------------------------------
-----------------------------------"
#to calculate runtime for 100 simuations of GA
start_time = Sys.time()
for (i in 1:100) {

  GA <- ga(type = "permutation", fitness = tspFitness, distMatrix = dis
t_mat, lower = 1,
           upper = attr(dist_mat, "dim"), popSize = 10, maxiter = 500,
run = 100, pmutation = 0.2,
           monitor = NULL, keepBest = TRUE)
}
end_time = Sys.time()
runtime = end_time-start_time
runtime

"-----------------------------------------------------------------------
------------------------------------"
#to calculate mean runtime
start_time1 = c()
end_time1 = c()
for (i in 1:100) {
  start_time1 = c(start_time1,Sys.time())
  GA <- ga(type = "permutation", fitness = tspFitness, distMatrix = dis
t_mat, lower = 1,
           upper = attr(dist_mat, "dim"), popSize = 10, maxiter = 500,
run = 100, pmutation = 0.2,
           monitor = NULL, keepBest = TRUE)
  end_time1 = c(end_time1,Sys.time())
}
runtime1 = end_time1 - start_time1
mean(runtime1)


"-----------------------------------------------------------------------
------------------------------------"
#plotting runtime for 100 simulations
plot(runtime1,type="l")


"-----------------------------------------------------------------------
-----------------------------------"
n.iterations = GA@iter # number of iterations in GA
distances = numeric(n.iterations) # initializing zeros matrix to collec
t distances

# looping through the best solution of every iteration and storing the
distance for that solution
for (i in 1:n.iterations){
  distances[i] = distance(c(as.vector(GA@bestSol[[i]]), GA@bestSol[[i]]
[1,1]))
}



# collecting the distances in a dataframe for plotting
results <- data.frame(Iteration = seq(1:n.iterations), Distance = dista
nces)

library(ggplot2)

# Plot for Iteration vs Distance of Best Solution
ggplot(data = results, aes(x = Iteration, y = Distance))+
  geom_line(color = "#00AFBB", size = 1) + theme_minimal() +
  ggtitle("Distance for the Best Solution After Every Iteration")
"-----------------------------------------------------------------------
----------------------------"
mds <- cmdscale(dist_mat_2)
x <- mds[, 1]
y <- -mds[, 2]
n <- length(x)
B <- 100
fitnessMat <- matrix(0, B, 2)
A <- matrix(0, n, n)
```

```r
for (b in seq(1, B)) {
  # run a GA algorithm
  GA.rep <- ga(type = "permutation", fitness = tspFitness, distMatrix =
 dist_mat,
                 lower = 1, upper = attr(dist_mat,"dim"), popSize = 10, m
axiter = 500, run = 100,
                 pmutation = 0.2, monitor = NULL)

  tour <- GA.rep@solution[1, ]
  tour <- c(tour, tour[1])
  fitnessMat[b, 1] <- max(GA.rep@fitness)
  fitnessMat[b, 2] <- mean(GA.rep@fitness)
  A <- A + getAdj(tour)
}


plot.tour <- function(x, y, A) {
  n <- nrow(A)
  for (ii in seq(2, n)) {
    for (jj in seq(1, ii)) {
      w <- A[ii, jj]
      if (w > 0)
        lines(x[c(ii, jj)], y[c(ii, jj)], lwd = w, col = "lightgray")
    }
  }
}


plot(GA, main = "GA progression")
points(rep(50, B), fitnessMat[, 1], pch = 16, col = "lightgrey")
points(rep(55, B), fitnessMat[, 2], pch = 17, col = "lightblue")
title(sub = "Best and Avg at 50th iteration over 100 simulations")
"------------------------------------------------------------------------
----------------------------------------------"



mds <- cmdscale(dist_mat)
x <- mds[, 1]
y <- -mds[, 2]
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "")
abline(h = pretty(range(x), 10), v = pretty(range(y), 10),
       col = "light gray")
tour <- GA@solution[1, ]
tour <- c(tour, tour[1])
n <- length(tour)
arrows(x[tour[-n]], y[tour[-n]], x[tour[-1]], y[tour[-1]],
       length = 0.15, angle = 25, col = "green", lwd = 2)
text(x, y, labels = row.names(dist_mat),cex=0.8)
```

```r
## for simulated annealing 3 things are used :
## distance
# temperature
# accepting probability

## distance function

distance <- function(sq) {  # Target function
  sq2 <- embed(sq, 2)
  sum(dist_mat[cbind(sq2[,2], sq2[,1])])
}


## swapping is an important idea in SA

swap <- function(sq) {
  sq<- as.array(unique(sq))
  points <- sample(1:length(sq),2)
  temp<- sq[points[1]]
  sq[points[1]] = sq[points[2]]
  sq[points[2]] = temp
  #return(sq)
  return(append(sq,sq[1]))
}

## accepting probabiility

acceptingProb <- function(delta_dist,temperature) {
  return(1/(1+exp(delta_dist/temperature)))
}

acProb2 <- function(delta_dist,temperature) {
  return(exp(-delta_dist/temperature))

}

temperature <- 100000000000
cities<- 10
coolingRate<- 0.01
sequence <- sample(x = 1:cities)
sequence <-append(sequence,sequence[1])
Distance <- distance(sq= sequence)
iteration <- 1
error <- c(Distance)

## keeping track of time # system time
startTime <- proc.time()
```

```r
while(proc.time() - startTime < 2) {
  ## changing the sequence
  newSequence <- swap(sq = sequence)
  newDistance<- distance(sq = newSequence)

  #random probability is need to get out of local minima
  rand <- runif(1, min = 0, max = 1)
  #accept <- acProb2(newDistance -Distance ,temperature)
  accept <- acceptingProb(newDistance -Distance , temperature)

  if(rand <= accept) {
    deltaDist <- newDistance - Distance
    cat("iteration: ", iteration, "Temperature: ", temperature, "\n")
    cat("sequence: ", sequence, ", New sequence: ", newSequence, "\n")
    cat("Total Distance: ", Distance, "Delta Distance: ", deltaDist, ",
 New Total Distance: ", newDistance, "\n")
    cat("\n\n")

    Distance <- newDistance
    sequence <- newSequence
    temperature <- temperature * (1-coolingRate)
    iteration <- iteration + 1
    error[iteration] = Distance
  }
}

plot(1:iteration, error,xlab = "Iterations", ylab = "Total Distance", m
ain = 'Total Distance vs Iterations', type = "l")
```