# Aegis School of Data Science Final Project

# Route Optimization

# By

# Rachuri Raviteja

# Santosh Namala

# John Mohd Bhat

## Project Guide:

## Vinay Kulkarni

## Minta Thomas

## 

## 

## Index

Index 2

Abstract 3

Introduction 4

Method 7

DATA 12

Results 13

Stage:1 – load the data 13

Stage:2 – Imputation of NA values 14

Stage:3 – Creating the initial Population 15

Stage:4 – Finding the fitness of the initial population 16

Stage:5 – Running the genetic algorithm 17

Appendices: Source Code 21

References 32

## 

## Abstract

This paper presents a case study of a company “Spring-Cleaning” who are trying to optimize the vehicle routes in order to efficiently satisfy their customers. This will be accomplished by designing a solution using advanced optimization techniques, transforming this problem in a typical Vehicle Routing Problem where the goal , given a set of locations, duration of time between the cities, and duration of time to complete the work at the city; is to find the best route.The result is an algorithm which provides the optimal routes in order to service requests by dynamically scheduling their resources under various constraints .

Problem Statement

Spring-Cleaning offers a wide variety of cleaning services. They provide carpet cleaning, tile and grout cleaning, upholstery cleaning, hardwood floor cleaning and air duct cleaning, etc. Broadly, they are categorized as residential and commercial cleaning. They have presence in 48 cities nationwide with various depots servicing specific regions.

Spring-Cleaning currently has an in house software that collects and maintains the services and schedules and decides vehicle routes. You need to work with Spring-Cleaning to develop an optimal routing solution for their vehicles. This will be accomplished by designing a solution using advanced optimization techniques. Spring-Cleaning strives to service requests by dynamically scheduling their resources under various constraints. You will be given data with routes scheduled on one day for each of their depots. Every vehicle in a depot would have a set of stops for the day, where stops means customers. In addition, they have to consider constraints like length of the time to complete the job, distance between stops and if the customers have specified a time when they would like to get the service done, etc.

## Introduction

This problem of route optimization is also called as the Vehicle Routing Problem or VRP for short. The vehicle routing problem (VRP) is a [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization) and [integer programming](https://en.wikipedia.org/wiki/Integer_programming) problem which asks "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?". It generalises the well-known [travelling salesman problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem) (TSP). It first appeared in a paper by [George Danzig](https://en.wikipedia.org/wiki/George_Dantzig) and John Ramser in 1959,in which first algorithmic approach was written and was applied to petrol deliveries. Often, the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. The objective of the VRP is to minimize the total route cost. In 1964, Clarke and Wright improved on Danzig and Ramser's approach using an effective greedy approach called the savings algorithm.

Determining the optimal solution is an [NP-hard](https://en.wikipedia.org/wiki/NP-hard) problem in [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization), so the size of problems that can be solved optimally is limited. The commercial solvers therefore tend to use heuristics due to the size of real world VRPs and the frequency that they may have to be solved.

The VRP concerns the service of a delivery company. How things are delivered from one or more depots which has a given set of home vehicles and operated by a set of drivers who can move on a given road network to a set of customers. It asks for a determination of a set of routes, S, (one route for each vehicle that must start and finish at its own depot) such that all customers' requirements and operational constraints are satisfied and the global transportation cost is minimized. This cost may be monetary, distance or otherwise.

The road network can be described using a [graph](https://en.wikipedia.org/wiki/Graph_%28discrete_mathematics%29) where the [arcs](https://en.wikipedia.org/wiki/Directed_edge) are roads and vertices are junctions between them. The arcs may be directed or undirected due to the possible presence of one way streets or different costs in each direction. Each arc has an associated cost which is generally its length or travel time which may be dependent on vehicle type.

To know the global cost of each route, the travel cost and the travel time between each customer and the depot must be known. To do this our original graph is transformed into one where the vertices are the customers and depot and the arcs are the roads between them. The cost on each arc is the lowest cost between the two points on the original road network. This is easy to do as [shortest path problems](https://en.wikipedia.org/wiki/Shortest_path_problems) are relatively easy to solve. This transforms the sparse original graph into a [complete graph](https://en.wikipedia.org/wiki/Complete_graph). For each pair of vertices i and j, there exists an arc (i,j) of the complete graph whose cost is written as _{ij} and is defined to be the cost of shortest path from i to j. The travel time _{ij} is the sum of the travel times of the arcs on the shortest path from i to j on the original road graph.

Sometimes it is impossible to satisfy all of a customer's demands and in such cases solvers may reduce some customers' demands or leave some customers unserved. To deal with these situations a priority variable for each customer can be introduced or associated penalties for the partial or lack of service for each customer given.

The objective function of a VRP can be very different depending on the particular application of the result but a few of the more common objectives are:

* Minimize the global transportation cost based on the global distance travelled as well as the fixed costs associated with the used vehicles and drivers
* Minimize the number of vehicles needed to serve all customers
* Least variation in travel time and vehicle load
* Minimize penalties for low quality service

The VRP has many obvious applications in industry. In fact the use of computer optimization programs can give savings of 5% to a company as transportation is usually a significant component of the cost of a product (10%) - indeed the transportation sector makes up 10% of the [EU's](https://en.wikipedia.org/wiki/European_Union)[GDP](https://en.wikipedia.org/wiki/Gross_domestic_product). Consequently, any savings created by the VRP, even less than 5%, are significant.

Here, the most commonly used techniques for solving Vehicle Routing Problems are listed. Near all of them are heuristics and metaheuristics because no exact algorithm can be guaranteed to find optimal tours within reasonable computing time when the number of cities is large. This is due to the NP-Hardness of the problem. Next we can find a classification of the solution techniques we have considered.

Metaheuristic Searches:-

* [Genetic Algorithms](http://neo.lcc.uma.es/vrp/solution-methods/metaheuristics/genetic-algorithm/)

## Method

For this problem we will be using the Genetic Algorithm or GA. A genetic algorithm (GA) is great for finding solutions to complex search problems. They're often used in fields such as engineering to create incredibly high quality products thanks to their ability to search a through a huge combination of parameters to find the best match. For example, they can search through different combinations of materials and designs to find the perfect combination of both which could result in a stronger, lighter and overall, better final product. They can also be used to design computer algorithms, to schedule tasks, and to solve other optimization problems. Genetic algorithms are based on the process of evolution by natural selection which has been observed in nature. They essentially replicate the way in which life uses evolution to find solutions to real world problems. Surprisingly although genetic algorithms can be used to find solutions to incredibly complicated problems, they are themselves pretty simple to use and understand.

## How they work

As we now know they're based on the process of natural selection, this means they take the fundamental properties of natural selection and apply them to whatever problem it is we're trying to solve.  
  
The basic process for a genetic algorithm is:

1. Initialization - Create an initial population. This population is usually randomly generated and can be any desired size, from only a few individuals to thousands.
2. Evaluation - Each member of the population is then evaluated and we calculate a 'fitness' for that individual. The fitness value is calculated by how well it fits with our desired requirements. These requirements could be simple, 'faster algorithms are better', or more complex, 'stronger materials are better but they shouldn't be too heavy'.
3. Selection - We want to be constantly improving our populations overall fitness. Selection helps us to do this by discarding the bad designs and only keeping the best individuals in the population.  There are a few different selection methods but the basic idea is the same, make it more likely that fitter individuals will be selected for our next generation.
4. Crossover - During crossover we create new individuals by combining aspects of our selected individuals. We can think of this as mimicking how sex works in nature. The hope is that by combining certain traits from two or more individuals we will create an even 'fitter' offspring which will inherit the best traits from each of it's parents.
5. Mutation - We need to add a little bit randomness into our populations' genetics otherwise every combination of solutions we can create would be in our initial population. Mutation typically works by making very small changes at random to an individuals genome.
6. And repeat! - Now we have our next generation we can start again from step two until we reach a termination condition.

## Termination

There are a few reasons why you would want to terminate your genetic algorithm from continuing it's search for a solution. The most likely reason is that your algorithm has found a solution which is good enough and meets a predefined minimum criteria. Offer reasons for terminating could be constraints such as time or money.

## Limitations

## Imagine you were told to wear a blindfold then you were placed at the bottom of a hill with the instruction to find your way to the peak. You're only option is to set off climbing the hill until you notice you're no longer ascending anymore. At this point you might declare you've found the peak, but how would you know? In this situation because of your blindfolded you couldn't see if you're actually at the peak or just at the peak of smaller section of the hill. We call this a local optimum. Below is an example of how this local optimum might look: http://www.theprojectspot.com/images/post-assets/tutorials/ga1/localopt.jpg Unlike in our blindfolded hill climber, genetic algorithms can often escape from these local optimums if they are shallow enough. Although like our example we are often never able to guarantee that our genetic algorithm has found the global optimum solution to our problem. For more complex problems it is usually an unreasonable exception to find a global optimum, the best we can do is hope for is a close approximation of the optimal solution.

The method chosen to solve the problem is Genetic Algorithm because this provide a list of possible solutions and not a single optimal route. Since there are many clients who would require attention at the same time, this is the best possible solution. The Genetic algorithm was programmed in R.

The general flow chart for a Genatic Algorithm for the project:

Things to be noted in the algorithm are:

1. The sample space should consist of all possible solution, but the solutions should not be repeated.
2. Fitness functions should be calculated based on the constraints provided, in this case it is the total time taken to complete the route and the number of stops that can be made in a day.
3. We need to provide a good estimated probability for the crossover and mutation to occur.
4. Once a route is created, we need to be certain that no stop is repeated again.
5. The mutated children should be re-entered into the gene pool so as to provide better results
6. The number of simulations should be chosen such that, a near optimum value can be reached.

## DATA

Data for project consists of below csv files

1. JobExecutionTime.csv – which will provide you information about job execution at a particular Stop ID (Customer)
2. TravelTime.csv – which will provide you information about travel time from StopID\_1 to StopID\_2

The data type for each stop ID is in integer. It has to be converted to a character type to perform most of the functions.

The data is clean but the TravelTimes.csv file has information for only 68 Stop ids. The rest of the data has to be added using a suitable imputation process.

The following constraints were also provided.

1. The number of stops for each vehicle should be less than 8 including the Depots.
2. The total time in minutes should be less than 660 ie -11 hours

This algorithm is tailored only to work with one requirement ie time constraints. If there are other factors to take into account such as load, distance, type of service to be provided etc. the code would have to be changed.

## Results

### Stage:1 – load the data

Load the CSV into R.

Jobexecution: (First 6 rows)

|  |  |  |
| --- | --- | --- |
|  | StopId | EstimatedDurationMinutes |
| 1 | 12308520 | 180 |
| 2 | 12332932 | 30 |
| 3 | 12351217 | 75 |
| 4 | 12351082 | 90 |
| 5 | 12350656 | 30 |
| 6 | 12345328 | 135 |

TimeTravel: (First 6 rows)

|  |  |  |  |
| --- | --- | --- | --- |
|  | stopId1 | stopId2 | time |
| 1 | 12308520 | 12332932 | 36 |
| 2 | 12308520 | 12337737 | 18 |
| 3 | 12308520 | 12338526 | 26 |
| 4 | 12308520 | 12339545 | 34 |
| 5 | 12308520 | 12342376 | 19 |
| 6 | 12308520 | 12343415 | 53 |

It can be seen from the structure of both the data frames that the all the data is in Integer or factor form. The data for the stopIds are to be converted to character type in order to perform string operations such as concatenation, split etc.

Structure of Job Execution data frame:

data.frame': 68 obs. of 2 variables:

$ StopId : int 12308520 12332932 12351217 12351082 12350656 12345328 12345730 12348388 12347928 12339545 ...

$ EstimatedDurationMinutes: int 180 30 75 90 30 135 150 15 45 15 ...

Structure of Time Travel data frame:

'data.frame': 3782 obs. of 3 variables:

$ stopId1: Factor w/ 62 levels "12308520","12332932",..: 56 56 56 56 56 56 56 56 56 56 ...

$ stopId2: Factor w/ 62 levels "12308520","12332932",..: 16 26 28 7 60 36 9 18 50 49 ...

$ time : int 15 16 5 17 17 13 61 62 58 68 ...

Notice that the there are 7 stopID’s which are missing. There is no travel time to and from these stopIds as well therefore the data has to be imputed. The NA values cannot be dropped as it would effect the performance of the model.

### Stage:2 – Imputation of NA values

Using the function permutations from the package “gtools” create a list of all possible combinations of stop IDs

permutations(n, r, v=1:n, set=TRUE, repeats.allowed=FALSE)

n = Size of the source vector

r = Size of the target vectors

v = Source vector. Defaults to 1:n

The result of which would look like this: (First 6 rows)

|  |  |  |
| --- | --- | --- |
|  | stopId1 | stopId2 |
| 1 | 12308520 | 12332932 |
| 2 | 12308520 | 12337737 |
| 3 | 12308520 | 12338526 |
| 4 | 12308520 | 12339545 |
| 5 | 12308520 | 12339675 |
| 6 | 12308520 | 12341082 |

Using the merge function join the traveltimes by mapping them from TravelTime data frame. Replace the NA values with the max value for time travel. Since this is an optimization technique, worst case scenario has to be considered in order to achieve model with the highest reward. The output of this would look similar to the TravelTime data frame with the addition of any stopids which were missing earlier.

### Stage:3 – Creating the initial Population

Then the initial population is created by selecting a random StopID from the JobExecution Data frame. The output is a collection of routes which varies in number depending on the starting StopID.

The initial population would look like this:

|  |  |
| --- | --- |
|  | x |
| 1 | Depot-12347928-12350822-Depot |
| 2 | Depot-12351315-12351426-12348388-12348192-12351217-Depot |
| 3 | Depot-12341082-12350656-12337737-12347871-12350687-Depot |
| 4 | Depot-12345229-12350103-12343415-12350468-12308520-Depot |
| 5 | Depot-12345328-12346603-12347260-12350265-12348803-Depot |
| 6 | Depot-12350684-12349786-12350842-Depot |
| 7 | Depot-12350484-12332932-12351024-12349516-12349968-Depot |
| 8 | Depot-12350492-12338526-12351082-12349651-Depot |
| 9 | Depot-12339545-12350135-12351488-12351272-12348444-Depot |
| 10 | Depot-12342376-12351196-12349706-12350358-12350818-Depot |
| 11 | Depot-12350501-12350840-Depot |
| 12 | Depot-12349842-12348335-Depot |
| 13 | Depot-12348709-12350757-12348665-12350627-12349283-Depot |
| 14 | Depot-12339675-12348896-Depot |
| 15 | Depot-12350406-12349009-12350758-12348694-Depot |
| 16 | Depot-12345730-12351049-12349074-Depot |
| 17 | Depot-12346752-12349005-12351769-Depot |
| 18 | Depot-12349989-Depot |
| 19 | Depot-12349553-Depot |
| 20 | Depot-12349057-Depot |

### Stage:4 – Finding the fitness of the initial population

Then the fitness function for the population is calculated using the function totaltime. Which is the sum of the time taken at each stop and the time taken to travel from one stop to another.

The fitness function for the initial population =

> totaltime(InitialPopulation)

[1] 9287

### Stage:5 – Running the genetic algorithm

The genetic algorithm function is executed. It uses the crossover and the mutation functions to improve the fitness of the existing population depending on the number of iterations it generates a different optimum value

For 1000 iterations :

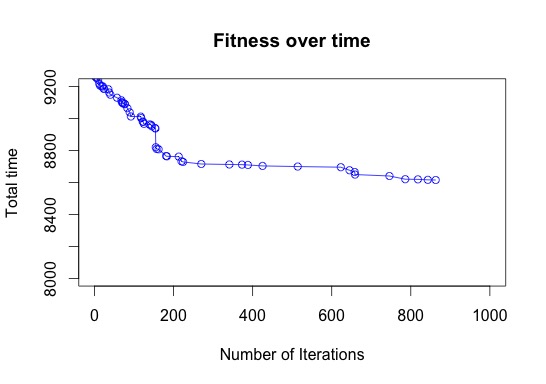
The new populations set

|  |  |
| --- | --- |
|  | x |
| 1 | Depot-12347928-12350822-Depot |
| 2 | Depot-12339545-12350135-12351488-12351272-12348444-Depot |
| 3 | Depot-12342376-12351196-12348896-Depot |
| 4 | Depot-12348709-12349005-12350484-Depot |
| 5 | Depot-12349842-12350684-12348694-Depot |
| 6 | Depot-12341082-12350406-12350358-Depot |
| 7 | Depot-12339675-12349706-12350103-12350842-Depot |
| 8 | Depot-12351315-12350627-12348665-Depot |
| 9 | Depot-12349786-12343415-12349516-12349968-Depot |
| 10 | Depot-12345229-12332932-12349989-Depot |
| 11 | Depot-12350687-12346603-12350840-Depot |
| 12 | Depot-12349057-12348192-12351217-Depot |
| 13 | Depot-12350501-12349283-12351082-12350758-Depot |
| 14 | Depot-12345730-12351426-12347260-12348335-Depot |
| 15 | Depot-12351049-12348388-12350757-12349651-Depot |
| 16 | Depot-12337737-12338526-12350656-12349074-Depot |
| 17 | Depot-12348803-12350818-12350492-12351024-Depot |
| 18 | Depot-12350265-12346752-12349009-Depot |
| 19 | Depot-12345328-12350468-12308520-Depot |
| 20 | Depot-12349553-12347871-12351769-Depot |

The new fitness function:

> totaltime(InitialPopulation)

[1] 8615



From the graph it can be seen that the transformation from initial fitness to the final fitness, it first decrease at frequent intervals. Then gradually it takes more and more iterations to generate a better fitness.

For 5000 Iterations:

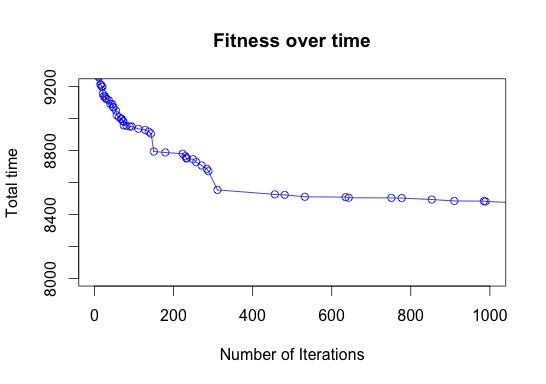
|  |  |
| --- | --- |
|  | x |
| 1 | Depot-12347928-12350822-Depot |
| 2 | Depot-12350501-12347260-12345730-Depot |
| 3 | Depot-12349989-12350468-12308520-Depot |
| 4 | Depot-12350492-12346752-12350627-Depot |
| 5 | Depot-12349005-12350757-12350656-12349057-Depot |
| 6 | Depot-12350842-12345229-12349968-Depot |
| 7 | Depot-12349516-12351217-12348192-Depot |
| 8 | Depot-12338526-12337737-12348803-12346603-Depot |
| 9 | Depot-12351082-12351426-12348335-12332932-Depot |
| 10 | Depot-12348896-12348709-12351769-Depot |
| 11 | Depot-12345328-12342376-12351024-Depot |
| 12 | Depot-12348388-12339675-12349786-12350840-Depot |
| 13 | Depot-12351272-12341082-12349074-12351488-12348444-Depot |
| 14 | Depot-12339545-12351049-12350818-12351315-Depot |
| 15 | Depot-12350684-12350103-12349842-12348665-Depot |
| 16 | Depot-12347871-12349283-12349706-12348694-Depot |
| 17 | Depot-12349651-12351196-12350135-Depot |
| 18 | Depot-12349009-12350265-12350358-Depot |
| 19 | Depot-12350758-12350484-12349553-Depot |
| 20 | Depot-12350687-12343415-12350406-Depot |

The fitness function:

> totaltime(InitialPopulation)

[1] 8336

The greater the number of iterations higher is the chance of reaching a better, more optimal value



**Summary**

There are other route optimization techniques which take into account other factors and produce a better result. But as a novice R programmer this is the best result that I could achieve.

With further knowledge of machine learning a more ergonomic and effective code could have been made.

## 

## Appendices: Source Code

## Genetic Algorithm:

## setwd('G:\\project\\np')

## ## Reading the Datasets

## Travel\_Time <- read.csv('TravelTime.csv', stringsAsFactors = F)

## Job\_Time <- read.csv('JobExecutionTime.csv', stringsAsFactors = F)

## ## Adding the Depot as one of the stops.

## ## We are assuming that the Job Execution time at depot is 0

## library(utils)

## Job\_Time[69,]<-c('Depot',0)

## ## Finding all the combinations of stops

## ## Example: If there are 4 stops (s1,s2,s3,s4), the below function

## ## will return [(s1,s2), (s1,s3), (s1,s4), (s2,s3), (s2,s4), (s3,s4)]

## comb <- combn(Job\_Time$StopId,2)

## ## Including all the permutations of the combinations list

## ## This will return [(s1,s2), (s1,s3), (s1,s4), (s2,s3), (s2,s4), (s3,s4)

## ## (s2,s1), (s3,s1), (s4,s1), (s3,s2), (s4,s2), (s4,s3)]

## comb <- data.frame('stopId1'=c(comb[1,],comb[2,]),'stopId2'=c(comb[2,],comb[1,]))

## ## Merging the Travel time Dataframe with the combinations. The combinations

## ## that are not present in Travel time will take NA values as time

## new\_df <- merge(comb, Travel\_Time, by=c('stopId1','stopId2'), all = T)

## ## The below code will impute the NA values in the dataframe with the

## ## maximum time of the starting point and ending point

## for (i in which(is.na(new\_df$time))){

## new\_df$time[i] <- max(na.omit(c(new\_df$time[new\_df$stopId1==new\_df$stopId1[i]],new\_df$time[new\_df$stopId2==new\_df$stopId2[i]])))

## }

## ## Combining the Job Execution time to the dataframe

## new\_df$Job\_Time <- sapply(new\_df$stopId2,function(x) Job\_Time$EstimatedDurationMinutes[Job\_Time$StopId==x])

## ## Type conversion

## new\_df$Job\_Time <- as.integer(new\_df$Job\_Time)

## ## Calculatimg the total time taken to travel and complete the job

## new\_df$Total\_Time <- new\_df$time+new\_df$Job\_Time

## ## setting the total number of stops

## no\_stops = nrow(Job\_Time)

## ## Initializing time for using in the loop

## time=0

## new\_df$stopId1<-as.character(new\_df$stopId1)

## new\_df$stopId2<-as.character(new\_df$stopId2)

## give\_route<-function(x){

## start=1

## i=1

## end=sample(5:7,1)

## route = list()

## while(start<69){

## route[[i]]<-c('Depot',x[start:end],'Depot')

## i=i+1

## print (start)

## print (end)

## start=end+1

## if(start>61){

## end = 68

## } else{

## end = start + sample(5:7,1)

## }

## }

## return (route)

## }

## calculate\_penalty<-function(x){

## penal<-ifelse(x>660,1,0)

## penal[x>660]<-as.integer((x[x>660]-660)/60)+1

## penal[penal==1]<-66

## penal[penal==2 | penal==3]<-132

## penal[penal>3] <- 198

## return (penal)

## }

## calculate\_time<-function(x){

## time=as.numeric()

## for(i in 1:length(x)){

## time[i]=0

## for(j in 2:length(x[[i]])){

## time[i]=time[i]+new\_df$Total\_Time[new\_df$stopId1==x[[i]][j-1] & new\_df$stopId2==x[[i]][j]]

## }

## }

## return (time+calculate\_penalty(time))

## }

## mutate<-function(x){

## number\_changes<-sample(1:(length(x)/2),1)

## number\_changes<-sample(1:length(x),number\_changes)

## for (i in number\_changes){

## if(length(x[[i]]<4)) next

## if(length(x[[i]]<6)){

## shuffles=2

## } else{

## shuffles = sample(2:(length(x[[i]])/2))

## }

## shuffles=sample(2:(length(x[[i]])-1),shuffles)

## x[[i]][shuffles]=sample(x[[i]][shuffles])

## }

## return (x)

## }

## crossover<-function(x){

## t=calculate\_time(x)

## c1=which.max(t)

## c2=which.min(t)

## 

## minim<-x[[c2]]

## maxim<-x[[c1]]

## 

## if(length(x[[c2]])>3){

## x[[c1]]=c(maxim[1:(length(maxim)/2)],

## minim[as.integer(length(minim)/2 +1):length(minim)])

## x[[c2]]=c(minim[1:(length(minim)/2)],

## maxim[as.integer(length(maxim)/2 +1):length(maxim)])

## }

## return (x)

## }

## create\_solution<-function(x){

## if(length(x)<8){

## return (list(x))

## }

## number\_genes = sample(5:7,1)

## chromosome = sample(1:length(x),number\_genes)

## soln = list(x[chromosome])

## x=x[-chromosome]

## soln = append(soln,create\_solution(x))

## return (soln)

## }

## create\_chromosomes<-function(x){

## soln=create\_solution(x)

## for (i in 1:length(soln)){

## soln[[i]]<-c('Depot',soln[[i]],'Depot')

## }

## return (soln)

## }

## final\_times=integer()

## final\_plot=integer()

## final\_solution = list()

## final\_time = 999999

## iter=0

## converge=0

## while(converge==0){

## iter=iter+1

## if(iter>5000){

## converge=1

## break

## }

## soln=create\_chromosomes(Job\_Time$StopId[1:68])

## time = calculate\_time(soln)

## if(sum(time)<sum(final\_time)){

## final\_time=time

## final\_times=append(final\_times,sum(final\_time))

## final\_solution=soln

## iter=0

## }

## 

## for(j in 1:100){

## iter=iter+1

## if(iter>5000){

## converge=1

## break

## }

## final\_plot=append(final\_plot,sum(final\_time))

## soln1 = mutate(soln)

## if(sum(calculate\_time(soln1))<sum(time)){

## time = sum(calculate\_time(soln1))

## soln=soln1

## if(sum(time)<sum(final\_time)){

## final\_time=time

## final\_times=append(final\_times,sum(final\_time))

## final\_solution=soln

## iter=0

## }

## }

## }

## for(j in 1:100){

## iter=iter+1

## if(iter>5000){

## converge=1

## break

## }

## final\_plot=append(final\_plot,sum(final\_time))

## soln1=crossover(soln)

## if(sum(calculate\_time(soln1))<sum(time)){

## time = sum(calculate\_time(soln1))

## soln=soln1

## if(sum(time)<sum(final\_time)){

## final\_time=time

## final\_times=append(final\_times,sum(final\_time))

## final\_solution=soln

## iter=0

## }

## }

## }

## final\_plot=append(final\_plot,sum(final\_time))

## print(iter)

## }

## plot.ts(final\_times)

## plot.ts(final\_plot)

**Simulated Annealing :**

**setwd('C:\\Users\\sharath\\Desktop\\RO')**

**## Reading the Datasets**

**Travel\_Time <- read.csv('TravelTime.csv', stringsAsFactors = F)**

**Job\_Time <- read.csv('JobExecutionTime.csv', stringsAsFactors = F)**

**## Adding the Depot as one of the stops.**

**## We are assuming that the Job Execution time at depot is 0**

**library(utils)**

**Job\_Time[69,]<-c('Depot',0)**

**## Finding all the combinations of stops**

**## Example: If there are 4 stops (s1,s2,s3,s4), the below function**

**## will return [(s1,s2), (s1,s3), (s1,s4), (s2,s3), (s2,s4), (s3,s4)]**

**comb <- combn(Job\_Time$StopId,2)**

**## Including all the permutations of the combinations list**

**## This will return [(s1,s2), (s1,s3), (s1,s4), (s2,s3), (s2,s4), (s3,s4)**

**## (s2,s1), (s3,s1), (s4,s1), (s3,s2), (s4,s2), (s4,s3)]**

**comb <- data.frame('stopId1'=c(comb[1,],comb[2,]),'stopId2'=c(comb[2,],comb[1,]))**

**## Merging the Travel time Dataframe with the combinations. The combinations**

**## that are not present in Travel time will take NA values as time**

**new\_df <- merge(comb, Travel\_Time, by=c('stopId1','stopId2'), all = T)**

**## The below code will impute the NA values in the dataframe with the**

**## maximum time of the starting point and ending point**

**for (i in which(is.na(new\_df$time))){**

**new\_df$time[i] <- max(na.omit(c(new\_df$time[new\_df$stopId1==new\_df$stopId1[i]],new\_df$time[new\_df$stopId2==new\_df$stopId2[i]])))**

**}**

**## Combining the Job Execution time to the dataframe**

**new\_df$Job\_Time <- sapply(new\_df$stopId2,function(x) Job\_Time$EstimatedDurationMinutes[Job\_Time$StopId==x])**

**## Type conversion**

**new\_df$Job\_Time <- as.integer(new\_df$Job\_Time)**

**## Calculatimg the total time taken to travel and complete the job**

**new\_df$Total\_Time <- new\_df$time+new\_df$Job\_Time**

**## setting the total number of stops**

**no\_stops = nrow(Job\_Time)**

**## Initializing time for using in the loop**

**time=0**

**new\_df$stopId1<-as.character(new\_df$stopId1)**

**new\_df$stopId2<-as.character(new\_df$stopId2)**

**give\_route<-function(x){**

**start=1**

**i=1**

**end=sample(5:7,1)**

**route = list()**

**while(start<69){**

**route[[i]]<-c('Depot',x[start:end],'Depot')**

**i=i+1**

**#print (start)**

**#print (end)**

**start=end+1**

**if(start>61){**

**end = 68**

**} else{**

**end = start + sample(4:6,1)**

**}**

**}**

**return (route)**

**}**

**calculate\_penalty<-function(x){**

**penal<-ifelse(x>660,1,0)**

**penal[x>660]<-as.integer((x[x>660]-660)/60)+1**

**penal[penal==1]<-66**

**penal[penal==2 | penal==3]<-132**

**penal[penal>3] <- 198**

**return (penal)**

**}**

**calculate\_time<-function(x){**

**time=as.numeric()**

**for(i in 1:length(x)){**

**time[i]=0**

**for(j in 2:length(x[[i]])){**

**time[i]=time[i]+new\_df$Total\_Time[new\_df$stopId1==x[[i]][j-1] & new\_df$stopId2==x[[i]][j]]**

**}**

**}**

**return (time)**

**}**

**shuffle<-function(x){**

**number\_shuffles<-sample(2:10,1)**

**replace\_shuffles<-sample(1:68,number\_shuffles)**

**replaced\_shuffles<-sample(replace\_shuffles)**

**temp<-x[replace\_shuffles]**

**x[replace\_shuffles]=x[replaced\_shuffles]**

**x[replaced\_shuffles]=temp**

**return (x)**

**}**

**final\_solution = list()**

**final\_time = 999999**

**iter=1**

**sol\_times=as.integer()**

**times = as.integer()**

**final\_solns=as.integer()**

**for(i in 1:20){**

**arrangement\_m = Job\_Time$StopId[sample(1:68)]**

**arrangement<-arrangement\_m**

**time\_m=999999**

**iter=0**

**for(j in 1:200){**

**iter=iter+1**

**solution = give\_route(arrangement)**

**time = calculate\_time(solution)**

**time = time + calculate\_penalty(time)**

**times<-append(times,sum(time))**

**if(sum(time)<time\_m){**

**time\_m = sum(time)**

**sol\_times<-append(sol\_times,time\_m)**

**arrangement\_m=arrangement**

**if(time\_m<final\_time){**

**final\_time = sum(time)**

**final\_solution = solution**

**final\_solns<-append(final\_solns,final\_time)**

**}**

**} else if(runif(1)>0.8\*\*iter){**

**arrangement\_m=arrangement**

**time\_m = sum(time)**

**sol\_times<-append(sol\_times,time\_m)**

**}**

**arrangement = shuffle(arrangement\_m)**

**}**

**}**

**print (final\_time)**

**plot.ts(times)**

**plot.ts(sol\_times)**

**plot.ts(final\_solns)**

## References

1. Wikipedia – for the introduction to Vehicle Route Optimization
2. <http://neo.lcc.uma.es/vrp/solution-methods/> - for solution methods of VRP
3. Introduction to genetic algorithm - http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3