

Machine Learning 1
ICS2207
Assignment Report

Marc Ferriggi (286397M)

April 15, 2018

Contents

1	Introduction	1
2	Genetic Algorithms	2
3	Ant-Colony Optimisation	4
4	Results and Comparisons	6
5	Conclusion	8
6	Statement of Completion	8
7	Plagiarism Declaration Form	8
8	Appendix	8

1 Introduction

The Travelling Salesman Problem is a very common problem in the field of operations research. This has been studied extensively by mathematicians, computer scientists, and many great minds yet its complexity is still unknown [1]. The problem statement is given as follows:

“Given a collection of cities and the cost of travel between each pair of them, the travelling salesman problem, or TSP for short, is to find the

cheapest way of visiting all of the cities and returning to your starting point.”
[1]

Traditional methods for solving this problem come in three types; calculus based methods, exhaustive search methods and random search methods [3]. These methods, however, pose a large number of problems such as the algorithm converging to a local optimum or running in exponential time. Machine Learning Algorithms such as Genetic Algorithms (GAs) or the Ant-Colony Optimisation method (ACO) can be used to find accurate approximations to the solutions to the TSP and other similar optimisation problems.

In order to solve this problem using a Genetic Algorithm (Section 2) and the ACO method (Section 3), a few assumptions were made. Firstly, it was assumed that the data provided will come from a symmetric instance of TSP, i.e. the distance d from city c_i to city c_j $d(c_i, c_j) = d(c_j, c_i) \forall i, j \in [1, n]$. Another assumption that’s being made is that the “closed” version of TSP will be solved for this task, i.e. the salesman will end in the city where he started.

In Section 2 of this paper, Genetic Algorithms shall be defined and a method to solve the TSP using such an algorithm will be proposed. Section 3 shall define Ant-Colony Optimisation and propose a method for solving the TSP using this method. Section 4 will then compare the outcomes of the two algorithms and these results shall be discussed in detail. All code used will be listed in the Appendix (Section 8).

2 Genetic Algorithms

Genetic algorithms are designed to simulate a biological process [2], thus most of the terminology refers to the algorithm’s biological counterpart. The components that make up GAs are as follows:

- an objective (or fitness) function
- a population of chromosomes
- a selection operator on the chromosomes

- a crossover function which produces a new generation of chromosomes
- a random mutation function

The objective function is the function that the algorithm is trying to optimise. In Genetic Algorithms, this is often referred to as a *fitness* function, this term is in fact taken from evolutionary theory [2]. For the TSP, the fitness function is the sum of the distances between the points, the TSP in fact deals with minimising this sum in order to be able to find the shortest path. Given that the data under study is given in coordinate format [4], the fitness function used for the TSP is given in Equation 1 [2].

$$D = \sum_{k=1}^n \sqrt{(x_{k+1} - x_k)^2 + (y_{k+1} - y_k)^2} \quad (1)$$

A chromosome refers to a value that will be considered as the candidate solution to the optimisation problem. In the case of the TSP, the chromosome could be a permutation of the cities which represent the order that the salesman visits each city. Historically, chromosomes were encoded as a bit string, however it would make more sense not to encode it in this way for the TSP, this however will require a change in the crossover and mutation functions [2].

The selection operator refers to the method used to select which chromosomes are to be chosen for reproduction. In general, a fitter chromosome should be more likely to be selected. In the case of TSP, the probability function must be changed slightly from the original definition since the objective of the TSP is to minimize the fitness function (cost) and not maximize it, thus care must be taken to reverse the probability function.

Once the “fittest” chromosomes are selected, the crossover operator is then used to “combine” them. This operator “resembles the biological crossing over and recombination of chromosomes to create two offspring” [2]. In the case of the TSP, special care needs to be taken when designing this function and the classical way of defining this operator will not work since each city needs to be visited only once [5]. The technique which shall be used in the implementation for the TSP is known as the “cycle” crossover (as taken from [2]) and works as follows:

1. Initially, a random location is chosen in the length of the chromosome.

2. The two parent chromosomes exchange integers at this point to create the offspring.
3. If the integers are the same value then the offspring is the same as the parent and the algorithm terminates.
4. Otherwise, each offspring now has a duplicate integer, so switch the duplicate integer in the first offspring with the integer in the same location in the second offspring.
5. Repeat the above step until there are no duplicates in the first offspring (and thus the second offspring).
6. Both are now valid permutations.

The importance of the mutation step is to reduce the probability of the algorithm converging to a local optimum [5]. This is achieved by causing the algorithm to maintain diversity in the population, however it can cause it to converge more slowly [2]. Again, since the chromosome for the TSP is not encoded as a bit string, the mutation operator changes slightly from that described by Haupt in the original definition of a Genetic Algorithm. The implemented mutation operator randomly chooses two integers in a chromosome from the new generation and swaps them [2]. This will happen with a rather low probability in order to increase the rate of convergence.

3 Ant-Colony Optimisation

The Ant-colony Optimisation algorithm forms part of a class of population-based algorithms known as swarm intelligence (SI) that considers the collective behaviour of the population and individual solutions [6]. A general ACO algorithm has the following form [7]:

```

Set parameters , initialize pheromone trails
SCHEDULE ACTIVITIES
    ConstructAntSolutions
    DaemonActions {optional}
    UpdatePheromones
END_SCHEDULE_ACTIVITIES

```

The parameters initialised for my implementation of the Ant Colony Optimisation Algorithm are as follows:

1. no_of_ants: the number of ants in each iteration.
2. max_iter: the maximum number of iterations.
3. evaporation_rate: the rate at which the pheromones evaporate.
4. alpha: a parameter for calculating the probability of an ant selecting a certain route. Alpha gives more weighting to the pheromone values.
5. beta: a parameter for calculating the probability of an ant selecting a certain route. Beta gives more weighting to the heuristic value.
6. q0: this parameter gives the probability of using the previous ants' experience over selecting a random path.

Since the task at hand is to write an algorithm to solve the symmetric TSP and R is optimised for working with matrix algebra, the distances between each city were calculated as an initialisation step in order to avoid having to compute these values each time on every ant run. The distances between each city were stored in a symmetric matrix thus only the lower triangular matrix needed to be worked out using for loops.

In order to construct the ant solutions, initially, each ant is placed at random on a city. At each step, the ant then chooses the next city based on the probability matrix given in Equation 2.

$$p_{ij}^k(t) = \frac{[\tau_{ij}(t)]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} \text{ if } j \in \mathcal{N}_i^k \quad (2)$$

Where $p_{ij}^k(t)$ is the probability that ant k goes to city j from city i at iteration t , $\tau_{ij}(t)$ is the pheromone value on edge ij at iteration t , η_{ij} is the heuristic value on edge ij and \mathcal{N}_i^k is the set of all possible nodes reachable by ant k from node i which in our case will be the set of all possible nodes not yet visited by ant k [8].

The ant with the best solution after each iteration will then update the

pheromone values of each edge and is defined as seen in Equation 3 [6].

$$\begin{aligned}\tau_{ij} &= (1 - \rho)\tau_{ij} + \rho \sum_{k=1}^m \Delta\tau_{ij}^k, \\ \Delta\tau_{ij}^k &= \frac{1}{L^k}\end{aligned}\tag{3}$$

Where m is the number of ants, and L^k is the length of the best tour of that iteration. All pheromone values also evaporate at a rate defined by the parameter after each iteration.

4 Results and Comparisons

In general, when comparing the performance of the ACO Algorithm to the GA algorithm, one can note that the ACO algorithm seems to find a better approximation to the solution than the GA, and it does it in much less time using less iterations. This section compares the results of both algorithms on specific instances. By simply looking at the plots of the standard error of our estimated value compared with the iteration number, one can note the significant difference in performance. Note that with very large numbers, the Genetic Algorithm would take a very long time to compute and thus all the points were normalised between 0 and 1 in order to improve the speed of the algorithm. This step was then also repeated in the ACO Algorithm in order to be able to accurately compare the 2 methods.

Instance name: bays29.tsp

GA method: *see listing 1*

ACO method: *see listing 2*

GA Performance:

Time Taken: 177127.73ms

Best Distance Found: 6.129391

Actual Optimal Distance: 4.581227

Standard Error: 2.965736

ACO Performance:

Time Taken: 7170.33ms

Best Distance Found: 4.601584

Actual Optimal Distance: 4.581227

Standard Error: 0.000414

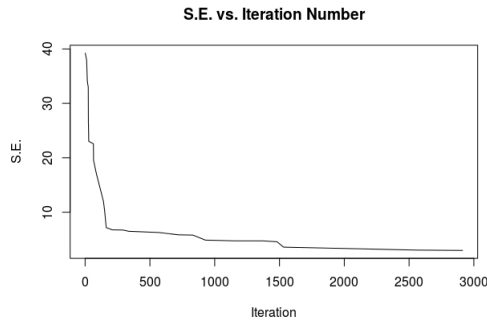


Figure 1: Genetic Algorithm

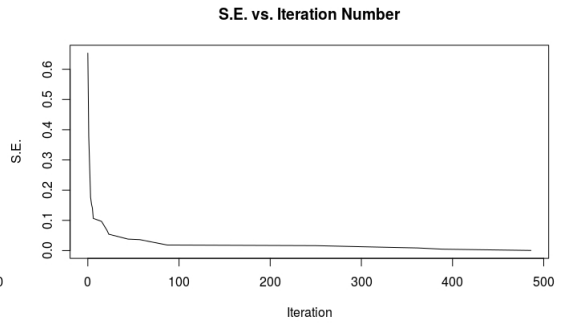


Figure 2: ACO Algorithm

Instance name: att48.tsp

GA method: see listing 1

ACO method: see listing 2

GA Performance:

Time Taken: 475314.39ms

Best Distance Found: 8.432387

Actual Optimal Distance: 5.377854

Standard Error: 9.330172

ACO Performance:

Time Taken: 49082.17ms

Best Distance Found: 5.431998

Actual Optimal Distance: 5.377854

Standard Error: 0.00101

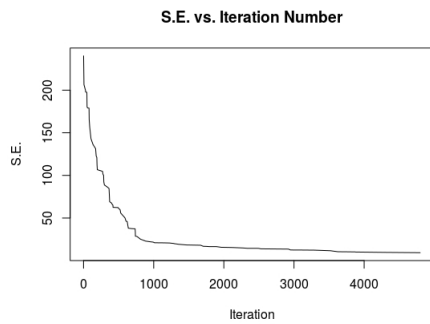


Figure 3: Genetic Algorithm

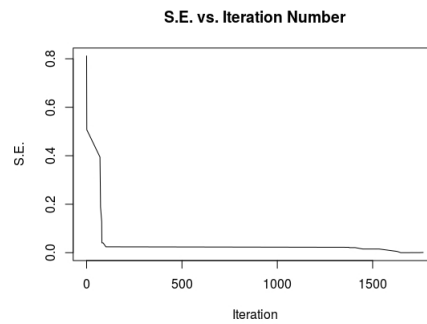


Figure 4: ACO Algorithm

Instance name: ch130.tsp

GA method: see listing 1

ACO method: see listing 2

GA Performance:

Time Taken: 1292756.23ms

Best Distance Found: 26.76519

Actual Optimal Distance: 8.703517

Standard Error: 326.2241

ACO Performance:

Time Taken: 57314.04ms

Best Distance Found: 9.542707

Actual Optimal Distance: 8.703517

Standard Error: 0.5555441

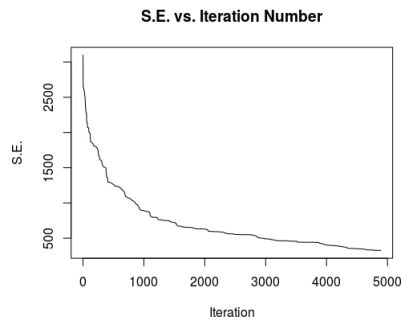


Figure 5: Genetic Algorithm

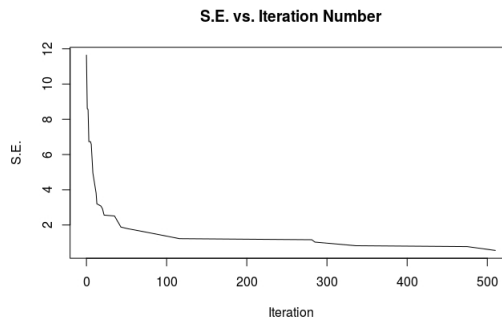


Figure 6: ACO Algorithm

5 Conclusion

6 Statement of Completion

7 Plagiarism Declaration Form

8 Appendix

```
1  rm(list=ls()) #Clear Environment
2
3  require(readxl) #needed to import data
4  Data = read_xlsx(file.choose(),col_names=FALSE) #
      Choose the data file you want to open
5  X = Data[2] #X coordinates indexed by city
6  Y = Data[3] #Y coordinats indexed by city
7  rm(Data) #remove Data (not needed)
8
9  #Normalize points to be between 0 and 1
10 #This is an optimisation step to improve the speed
    of the function
```



```

11  X = X/max(X)
12  Y = Y/max(Y)
13
14  #fitnessFunction:
15  #takes a population of chromosomes in Matrix form as
    an argument
16  #returns a vector of distances each element
    corresponding to a chromosome
17  #Example of use:
18  #a = c(1,2,3,4,5,6,7)
19  #b = c(2,4,3,1,5,15,12)
20  #c = c(10,2,3,1,5,6,11)
21  #pop = t(cbind(a,b,c))
22  #distances=fitnessFunction(pop)
23  fitnessFunction <- function(pop) {
24    Npop = nrow(pop)#size of population
25    Ncity = ncol(pop)#no. of cities
26    #append first city to the end in order to be able
    to calculate the distance
27    tour = cbind(pop,pop[,1])
28    #Empty vector to store distances of each
    chromosome
29    distances = mat.or.vec(Npop,1)
30
31    #loop over every tour in the population
32    for (i in 1:Npop) {
33      chromosome = tour[i,]
34      #for every city in the chromosome
35      distance = 0
36      for (j in 1:Ncity){
37        #fitness function as defined in Equation 1 of
    my Documentation
38        temp = sqrt((X[chromosome[j+1],1]-X[
    chromosome[j],1])^2+(Y[chromosome[j+1],1]-Y[
    chromosome[j],1])^2))
39        distance = distance + temp

```

```

40     }
41     distances[i]=distance
42 }
43 distances
44 }
45
46 #mate function , mates 2 chromosomes given a pointer
47   as
48 #the starting pointer and returns a list of the 2
49   new
50 #child chromosomes
51 mate <- function(matel ,mate2 ,pointer) {
52   temp = matel
53   change = TRUE
54   #loop until all values are unique
55   while (change) {
56     #swap values
57     matel[pointer]=mate2[pointer]
58     mate2[pointer]=temp[pointer]
59     #returns an array of indices which match the
60     value at matel[pointer]:
61     pointers = which(matel==matel[pointer] ,arr.ind =
62     TRUE)
63     #check if there exists an duplicate in the
64     chromosome
65     change = FALSE
66     tempPointer = pointer
67     for (j in 1:length(pointers)) {
68       if (pointers[j] !=pointer) {
69         #if there's a duplicate , point to it
70         tempPointer = pointers[j]
71         change = TRUE
72       }
73     }
74     pointer = tempPointer
75   }

```

```

71     return(unname(rbind(mate1, mate2)))
72 }
73
74 #function that mutates the chromosome
75 mutate <- function(chromosome) {
76     indices = sample(1:length(chromosome), 2, replace=
77     FALSE) #choose 2 random indices in the chromosome
78     #swap the elements
79     swapTemp = chromosome[indices[1]]
80     chromosome[indices[1]] = chromosome[indices[2]]
81     chromosome[indices[2]] = swapTemp
82     return(chromosome)
83 }
84
85 #Main Function:
86 #Initialize Genetic Algorithm Parametes:
87 NoOfCities = nrow(X) #Number of Cities
88 popSize = 10 #no of chromosomes in each population
89 pop = mat.or.vec(popSize, NoOfCities)
90 pop2 = mat.or.vec(popSize, NoOfCities)
91 best = mat.or.vec(1, NoOfCities) #vector to keep the
92     best chromosome so far
93 keep = 6 #no of chromosomes to be chosen as parents
94 mutationRate = 0.4 #probability of mutation
95 #noMutations = ceiling((popSize-1)*mutationRate) #
96     total number of mutations
97 Matings = ceiling((popSize-keep)/2) #number of
98     matings
99 maxit = 5000 #maximum number of iterations
100
101 #optTour = c
102     (1, 28, 6, 12, 9, 5, 26, 29, 3, 2, 20, 10, 4, 15, 18, 17, 14, 22, 11, 19, 25, 7, 23, 27, 8, 24,
103     #bays29
104     #optTour = c
105     (1, 8, 38, 31, 44, 18, 7, 28, 6, 37, 19, 27, 17, 43, 30, 36, 46, 33, 20, 47, 21, 32, 39, 48, 5

```

```

#att48
100 optTour = c
      (1,41,39,117,112,115,28,62,105,128,16,45,5,11,76,109,61,129,124,64,69,
101
      79,87,12,81,103,77,94,89,110,98,68,63,48,25,113,32,36,84,119,111,123,1
102
      10,14,67,13,96,122,55,60,51,42,44,93,37,22,47,40,23,33,21,126,121,78,6
#ch130
103 optDistance = 0
104 for (i in 2:NoOfCities) {
105     optDistance = optDistance + sqrt((X[optTour[i
      -1],1]-X[optTour[i],1])^2+(Y[optTour[i-1],1]-Y[
      optTour[i],1])^2)
106 }
107
108 bestVal = 1e+22 #Arbitrary large number
109 se = (bestVal-optDistance)^2
110 iteration = 0
111
112 #chromosomes that will survive and mate:
113 kept = mat.or.vec(keep,NoOfCities)
114 #stores the probability of each chromosome to
      survive
115 prob = mat.or.vec(popSize,1)
116
117 #populate initial population with random chromosomes
      :
118 for (i in 1:popSize) {
119     pop[i,] = sample(1:NoOfCities,NoOfCities) #creates
      a chromosome
120 }
121
122 #set starting time:
123 start_time = as.numeric(Sys.time())*1000;

```

```

124 #MAIN LOOP:
125 for (gen in 1:maxit) {
126     #compute the fitness function on the population
127     Lengths = fitnessFunction(pop)
128     #Save the best solution
129     if (min(Lengths)<bestVal) {
130         best[1,] = pop[which.min(Lengths),]
131         bestVal = min(Lengths)
132         se = c(se,(bestVal-optDistance)^2) #standard
133         error for graph
134         iteration = c(iteration,gen) #iter number for
135         graph
136         end_time=as.numeric(Sys.time())*1000
137         print(bestVal)
138     }
139
140     #selection
141     total = sum(Lengths)
142     #Probability of each chromosome to be selected
143     for (i in 1:popSize) {
144         prob[i] = 1- (Lengths[i]/total)
145         #This gives a higher probability value to the
146         shortest lengths
147         #Thus we need to select the chromosomes with the
148         highest probability
149     }
150     #selects the elements of the population to be kept
151     based on the probability distribution
152     #as defined above
153     odds = sample(1:popSize,keep,replace=TRUE,prob =
154     prob)
155     #choose the chromosomes to be kept and store them
156     in the new population
157     #keep the best and second best solutions
158     pop2[1,]=pop[which.min(Lengths),]
159     pop2[2,]=pop[which(Lengths==sort(Lengths,

```

```

decreasing=TRUE) [popSize-1]) [1] ,]
153   #choose parents, we'll choose 3 mums and 3 dads,
to have 6 kids:
154   for (i in 1:keep) {
155       #keep a record of the parents kept:
156       kept[i,] = pop[odds[i] ,]
157   }
158
159   index = 3
160   while (index < 9) {
161       #mate1, mate2 are random integers between 1 and
keep (index)
162       mate1=ceiling(runif(1, min=0, max=keep-1))
163       mate2=ceiling(runif(1, min=0, max=keep-1))
164       pointer = ceiling(runif(1,0,NoOfCities)) #random
int between 1 and NoOfCities
165       children = mate(kept[mate1,] , kept[mate2,] ,
pointer) #call the mate function
166       #mutate with probability and save to population
167       if (runif(1)<=mutationRate) {
168           pop2[index,] = mutate(children[1,])
169       } else {
170           pop2[index,] = children[1,]
171       }
172       if (runif(1)<=mutationRate) {
173           pop2[index+1,] = mutate(children[2,])
174       } else {
175           pop2[index+1,] = children[2,]
176       }
177       index=index+2
178   }
179   #Randomly fill remaining part of population with
chromosomes
180   for (i in 9:10) {
181       pop2[i,] = sample(1:NoOfCities ,NoOfCities) #
creates a chromosome

```

```

182     }
183
184     #Print iteration number
185     if (gen%%100==0) {
186         print(gen)
187     }
188     pop = pop2
189 }
190 #compute the fitness function on the population
191 Lengths = fitnessFunction(pop)
192 #Save the best solution
193 if (min(Lengths)<bestVal) {
194     best[1,] = pop[which.min(Lengths) ,]
195     bestVal = min(Lengths)
196     se = c(se,(bestVal-optDistance)^2) #standard error
197     for graph
198         iteration = c(iteration ,gen) #iter number for
199         graph
200     end_time=as.numeric(Sys.time())*1000
201     print(bestVal)
202 }
203 paste("Time Taken: ",end_time-start_time)
204 print(best[1,])
205 plot(iteration[-1],se[-1],type="l",main="S.E. vs.
206     Iteration Number",xlab="Iteration",ylab="S.E.")

```

Listing 1: Genetic Algorithm Code

```

1  rm(list=ls()) #Clear Environment
2
3  require(readxl) #needed to import data
4  Data = read_xlsx(file.choose(),col_names=FALSE) #
5      Choose the data file you want to open
6  X = Data[2] #X coordinates indexed by city
7  Y = Data[3] #Y coordinats indexed by city
8  rm(Data) #remove Data (not needed)

```

```

8
9  #Normalize points to be between 0 and 1
10 #This is an optimisation step to improve the speed
    of the function
11 X = X/max(X)
12 Y = Y/max(Y)
13
14 #####
15 #Parameters# <- adjusting these would change the
    performance of the algorithm
16 #####
17 no_of_ants = 10 #no of ants in each iteration
18 max_iter = 3000 #maximum number of iterations
19 evaporation_rate = 0.15 #evaporation rate of
    pheromones
20 alpha = 1 #alpha and beta are paramenters for
    calculating the prob matrix
21 beta = 4
22 q0 = 0.6 #probability of using other ant's
    experience
23
24 #####
25 #Functions#
26 #####
27 #this function returns the reciprocal of a number,
    unless that number is 0
28 probValues <- function(number) {
29     if (number == 0) {
30         number
31     } else {
32         1/number
33     }
34 }
35
36 #function that accepts a matrix with the first
    column being the starting point of each ant and the

```



```

    other columns all being 0
37 #a Heuristic Matrix and the pheromone value Matrix
    Tau and returns tours of each ant.
38 setRoutes <- function(Routes, Heuristic, Tau) {
39     #for every ant:
40     for (i in (1:nrow(Routes))) {
41         Memory = mat.or.vec(ncol(Routes),1) #Set Ant's
memory to 0s
42         Score = (Tau^alpha)*(Heuristic)^beta #Calculate
score matrix
43         Memory[1] = Routes[i,1] #Add starting node to
memory
44         #Loop until all cities have been visited:
45         for (j in 2:ncol(Routes)) {
46             currentCity = Memory[j-1]
47             #set score to 0 for current city:
48             Score[,currentCity] = 0
49             ParticularScore = Score[currentCity,]
50             if (runif(1)<=q0) {
51                 #find the city with the largest score:
52                 Memory[j] = which.max(ParticularScore)
53             } else {
54                 #Choose the next node based on probability
55                 Probability = ParticularScore/sum(
ParticularScore)
56                 Memory[j] = sample(1:length(Probability),1,
prob=Probability)
57             }
58         }
59         Routes[i,] = Memory
60     }
61     Routes
62 }
63
64 #function that calculates the length of each tour
65 fitnessFunction <- function(Routes, Distances) {

```

```

66     sum = mat.or.vec(nrow(Routes),1)
67     #for each ant j
68     for (j in 1:nrow(Routes)) {
69         cities = Routes[j,]
70         #for each city visited
71         for (i in 1:(ncol(Routes)-1)) {
72             #add up the distances
73             sum[j] = sum[j] + Distances[cities[i],cities[i
+1]]
74         }
75         sum[j] = sum[j] + Distances[cities[i+1],cities
[1]]
76     }
77 sum
78 }
79
80 #function that updates pheromones after an iteration
81 updatePheromones <- function(path, length,
    evaporation_rate, Tau) {
82     for (i in 1:(length(path)-1)) {
83         Tau[path[i],path[i+1]] = (1-evaporation_rate)*
Tau[path[i],path[i+1]]+evaporation_rate*(length)
^(-1);
84         Tau[path[i+1],path[i]] = Tau[path[i],path[i+1]]
85     }
86     Tau
87 }
88
89 #####
90 #Other Variables#
91 #####
92 start_time=as.numeric(Sys.time())*1000;
93 no_of_cities = nrow(X)
94 Tau = matrix(0.00001,nrow = no_of_cities , ncol = no_
    of_cities) #initial pheromone matrix
95 starting_nodes = mat.or.vec(no_of_ants,1)

```

```

96     dontStop = TRUE
97     #generate distance between cities matrix
98     Distances = matrix(0,nrow = no_of_cities ,ncol = no_
        of_cities)
99     Heuristic = Distances
100     iter = 0
101     #Note that Distances is a symmetric matrix since we'
        re working out the Symetric TSP
102     #, thus in order to improve complexity first the
        lower triangular part is worked out,
103     #then its transpose is added to itself.
104     for (i in 2:no_of_cities) {
105         for (j in 1:i) {
106             Distances[i,j] = sqrt((X[i,1]-X[j,1])^2+(Y[i,1]-
        Y[j,1])^2)
107         }
108     }
109     Heuristic = apply(Distances ,c(1,2) ,probValues) #
        Genarate the Heuristic Matrix (see function
        probValues)
110     Distances = Distances + t(Distances)
111     Heuristic = Heuristic + t(Heuristic)
112
113     bestRoute = mat.or.vec(no_of_cities ,1)
114     bestLength = 1e27 #Arbitrarily large number
115
116     #optTour = c
        (1,28,6,12,9,5,26,29,3,2,20,10,4,15,18,17,14,22,11,19,25,7,23,27,8,24,
        #bays29
117     #optTour = c
        (1,8,38,31,44,18,7,28,6,37,19,27,17,43,30,36,46,33,20,47,21,32,39,48,5
        #att48
118     optTour = c
        (1,41,39,117,112,115,28,62,105,128,16,45,5,11,76,109,61,129,124,64,69,
119

```

```

79,87,12,81,103,77,94,89,110,98,68,63,48,25,113,32,36,84,119,111,123,1

120
    10,14,67,13,96,122,55,60,51,42,44,93,37,22,47,40,23,33,21,126,121,78,6
    #ch130
121 optDistance = 0
122 for (i in 2:no_of_cities) {
123     optDistance = optDistance + Distances[optTour[i
        -1],optTour[i]]
124 }
125 optDistance = optDistance + Distances[optTour[i],
    optTour[1]]
126 se = (bestLength-optDistance)^2
127 iteration = 0
128
129 #####
130 #Main Algorithm Loop#
131 #####
132 while (dontStop) {
133     #initialize each ant in a starting node
134     starting_nodes = sample(1:no_of_cities,no_of_ants,
        replace = TRUE)
135     routes = matrix(0,nrow = no_of_ants,ncol = no_of_
        cities) #routes will store the routes of the ants
136     routes[,1] = starting_nodes
137
138     #generate the routes for each ant
139     routes = setRoutes(routes,Heuristic,Tau)
140
141     #build a solution
142     lengths = mat.or.vec(no_of_ants,1)
143     lengths = fitnessFunction(routes, Distances)
144
145     #save best route
146     bestAntIndex = which.min(lengths)
147     if (lengths[bestAntIndex]<bestLength) {

```

```

148     bestLength = lengths[bestAntIndex]
149     bestRoute = routes[bestAntIndex,]
150     se = c(se,(bestLength-optDistance)^2) #standard
error for graph
151     iteration = c(iteration,iter) #iter number for
graph
152     end_time = as.numeric(Sys.time())*1000
153     print(bestLength)
154 }
155
156 #update pheromone values
157 Tau = updatePheromones(routes[bestAntIndex,],
lengths[bestAntIndex],evaporation_rate,Tau)
158
159 #Stopping Criteria:
160 iter = iter+1
161 if (iter%%10==0) {print(iter)}
162 dontStop = iter<=max_iter
163 }
164 paste("Time Taken: ",end_time-start_time)
165 print(bestRoute)
166 plot(iteration[-1],se[-1],type="l",main="S.E. vs.
Iteration Number",xlab="Iteration",ylab="S.E.")
167

```

Listing 2: ACO Code

References

- [1] uwaterloo. *The Problem*.
<http://www.math.uwaterloo.ca/tsp/problem/index.html>.
[6th March 2018].
- [2] Carr, J. *An Introduction to Genetic Algorithms*.
<https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf>.
[16th May 2014].
- [3] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*.
[1989].
- [4] Unknown. *MP-TESTDATA - The TSPLIB Symmetric Traveling Salesman Problem Instances*.
<http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html>
- [5] Haupt, R.L., & Haupt, S.E. (2004). *Practical Genetic Algorithms* (2nd ed.). Hoboken: Wiley.
- [6] Tseng, S.P., et. al. *A fast Ant Colony Optimization for traveling salesman problem*.
<https://ieeexplore-ieee-org.ejournals.um.edu.mt/document/5586153/?reload=true>.
[23rd July 2010].
- [7] Dorgio, M. *Ant colony optimization*.
http://www.scholarpedia.org/article/Ant_colony_optimization.
[2007].
- [8] Garg D., Shah S. *ANT COLONY OPTIMIZATION FOR SOLVING TRAVELING DALESMAN PROBLEM*. *International Journal of Computer Science and System Analysis*, vol. 5, no. 1, pp. 23-29, 2011.
available: <http://www.serialsjournals.com/serialjournalmanager/pdf/1330336909.pdf>