MCMC and the Travelling Salesman Problem: The Curse of Dimensionality

David Veitch
University of Toronto
david.veitch@mail.utoronto.ca

November 2019

Abstract

This paper analyzes how Simulated Annealing can be applied to the Travelling Salesman Problem. The effect of varying the proposal of new states, employing restarts, and increasing the dimension of the problem are investigated. The results of this analysis show Simulated Annealing is able to effectively find the optimal solution in low dimensions, but struggles to find the optimal solution in higher dimensions.

1 Introduction

The Traveling Salesman Problem (TSP), is a famous, and widely studied optimaization problem. The TSP is as follows: what is the shortest way to travel between a number of cities. What appears to be a relatively simple problem suffers greatly from the curse of dimensionality. Optimizing a route between four cities is easy, and could even be done by hand. On the other hand, optimizing a route between thirty cities is much more difficult, with approximately 2.65×10^{32} possible routes. Given the large number of potential routes, the best one may hope to do is to search over a smaller number of routes to find an approximately best route.

Markov Chain Monte Carlo techniques (MCMC) can be used to estimate difficult-to-compute quantities, and is particularly useful when the exact form of the function one is estimating is unknown. These techniques are generally used to generate samples from probability distributions. However, since most functions can be viewed as an unnormalized probability density function, MCMC can be used in optimization for general functions.

2 Simulated Annealing

2.1 Optimizing a Modified f

A MCMC technique which is particularly well-suited towards optimization is simulated annealing. This technique seeks to find the maximum of a given function f by gradually optimizing a modified f which

has more pronounced maxima. Specifically, given a cooling schedule $\{\tau_n\}$ such that $\tau_n \searrow 0$, at each step the algorithm tends towards the maximum of $f^{1/\tau}$

2.2 Generating Proposals

As with any MCMC algorithm, in Simulated Annealing one must define a Markov Chain $X_0, X_1, X_2, \ldots$. For each n a new Y_n is proposed based on the previous state X_{n-1}. This Y_n is then accepted with probability $\min\left[1, \frac{f(Y_n)^{1/\tau_n}}{f(X_{n-1})^{1/\tau_n}}\right]$. If Y_n is accepted than $X_n = Y_n$, and if it is not accepted $X_n = X_{n-1}$. Proposed values Y_n are more likely to be accepted if $f(Y_n)^{1/\tau_n}$ is large, therefore this algorithm graviates towards values which maximize f^{1/τ_n}. Also as τ_n decreases it becomes harder to accept proposals which decrease the value of the function and the Markov Chain settles on values which maximize f^{1/τ_n}, and therefore also maximize f itself.

One should note that in the general MCMC setup one must specify the Markov Chain to be irreducible (positive probability of eventually getting from anywhere to anywhere else) and aperiodic (no forced cycles) with stationary density π to ensure $\mathcal{L}(X_n) \to \pi$. An irreducible Markov Chain which is time reversible and includes a possibility of rejecting proposals fufills these requirements.

3 Problem Formulation

The Iowa Democratic Caucuses are widely seen as one of the most important stages of the Democractic Party presidential primaries [4]. Taking place on Monday February 3 2020, they are the first state to vote, and many candidates invest huge resources into this race in order to generate momentum into other states' primaries. It is a common sight to see candidates flood to state fairs, local restaurants, and community halls to convince voters they should be the leader of the Democratic party. For this optimization problem we imagine planning the itinerary of a hypothetical candidate who wishes to minimize their driving time while visiting the K most populous cities in Iowa. We add a constraint that the route begins and ends in Des Moines, since any candidate would likely fly into Des Moines International Airport to begin their trip.

4 Data Sources

Mileage charts, driving times, latitude/longitudes, and populations were obtained for cities in Iowa [1; 2]. It should be noted that the driving times were calculated using approximate road speeds, therefore the shortest straight-line route may not necessarily be the fastest. As well, the driving times between two cities may differ depending on direction (this is caused by one-way roads).

Figure 1: Map of Iowa's 280 Towns and Cities.

5 Simulated Annealing Algorithm

5.1 Reframing the Minimization Problem

In a typical simulated annealing problem one seeks the largest mode of a probabilty density π. In our problem we seek to minimize the function $f(x)^{1/\tau}$, where

$$f(\text{route}) = \sum_{i=1}^{K-1} \text{Driving Time Between Cities}(i, i+1).$$

We turn this minimization problem into a by defining f' as an unnormalized Boltzman distribution taking the form

$$f'(\text{route})^{1/\tau} = e^{-f(\text{route})/\tau}.$$

Obviously $f'(\text{route})^{1/\tau}$ is maximized when $f(\text{route})$ is minimized. For each run of the algorithm we select the route with the minimum time as the solution.

5.2 Generating Proposals

For K cities with no constraints on where one starts and finishes, a route is an ordering $(\text{City}_1, \ldots, \text{City}_K)$. To generate a new proposal the locations on the route of J cities are rearranged at random (note this includes the possibility that the proposal is equal to the current route).

	City_1	City_2	City_3	City_4	City_5
Initial Route	Des Moines	Cedar Rapids	Davenport	Sioux City	Iowa City
Proposed Route (J=2)	**Davenport**	Cedar Rapids	**Des Moines**	Sioux City	Iowa City
Proposed Route (J=3)	**Iowa City**	Cedar Rapids	Davenport	**Des Moines**	**Sioux City**

Table 1: Example of Proposed Routes Based on an Initial Route

5.3 Cooling Schedules

Three cooling schedules were tested. Where M is the number of iterations and τ_0 is the initial temperature these are defined as

1. Logarithmic Cooling - $\tau_n = \tau_0 / \log(1+n)$
2. Linear Cooling - $\tau_n = \tau_0 - \tau_0 \frac{n}{M+1}$
3. Geometric Cooling - $\tau_n = \tau_0 r^n, \;\; 0 < r < 1$

5.4 Restarts

Given the large number of potential routes, and the possibility of the algorithm converging to a local, and not global optimum, restarts were used as is suggested in the literature [3]. That is, the algorithm was run multiple times and only the best solution was kept. The main benefit of using these restarts is beginning at initial values which are significantly different from one another.

6 Results

The results below are shown for an algorithm using six restarts of 50,000 iterations each. After some trial and error a geometric cooling schedule was used with $r = 0.9999$ and $\tau_0 = 10$. The logarithmic cooling schedule was found to converge much to slowly to an optimal solution. In general the algorithm was able to make the most progress towards optimal solutions at temperatures much lower than τ_0 and the linear cooling schedule was suboptimal as it spent relatively few iterations at these low temperatures.

6.1 A Simple Example

To check if the algorithm is in fact able to find the optimal solution more efficiently than just checking all of the permuations of a route, we try finding the optimal route between 11 cities where the start and finish is in Des Moines, and the optimal time can be found via brute force. In total there are $10! = 3,628,800$ permutations relative to total iterations of the algorithm of $5 \times 50,000 = 250,000$). We see that the algorithm does successfully find the optimal route, and does not even require all six restarts to do so.

Restart Number	Optimal	1	2	3	4	5	6
Route Time	15.226	15.226	15.371	15.226	15.226	15.371	15.590

Table 2: Simulated Annealing Results vs. Optimal Solution for 11 Cities

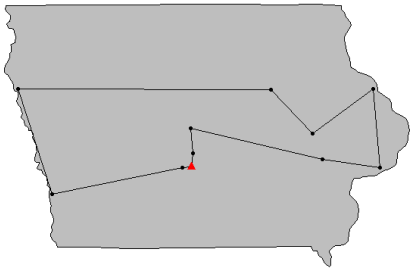


Figure 2: Optimal Route for 11 Cities with Start/Finish in Des Moines.

6.2 A Realistic Example - 30 Cities

Next we take a look at the algorithm's performance for 30 cities where the start and finish is in Des Moines. In this case the number of routes ($29! \approx 8.84 \times 10^{30}$) is far too large to check what the optimal solution is. It is also not initially obvious that for a randomly chosen initial route this algorithm will converge to anything resembling an optimal route. At each iteration we propose a new route by swapping two cities (other than the start and end city) at random.

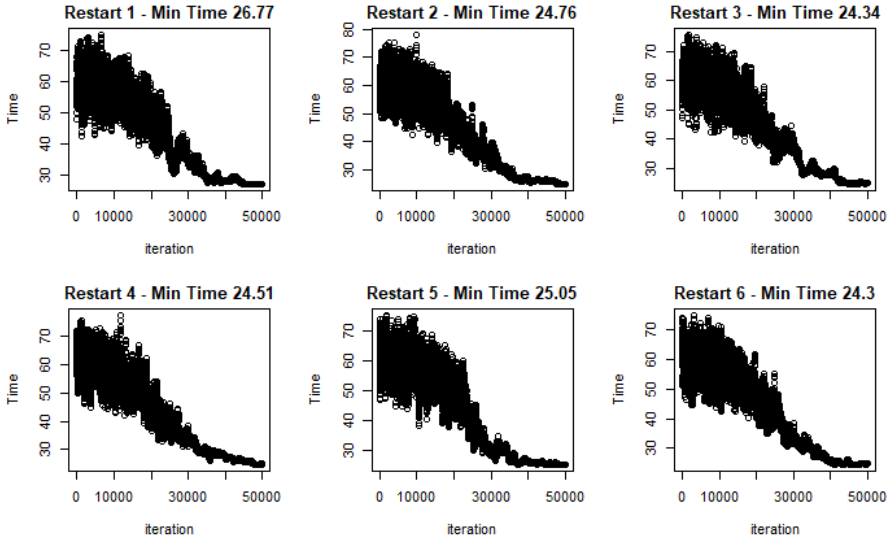


Figure 3: Progress of Algorithm for Finding Optimal Route for 30 Cities

As can be seen in Figure 3, for each restart the algorithm initially (i.e. at high temperatures) accepts many proposals, even ones that are much less optimal than the current state. As the temperature decreases the algorithm begins accepting only proposals with very similar, or better, times than the previous state. As can be seen in Figure 4 the algorithm begins by accepting roughly half of all proposals, and finishes by accepting less than 1% of proposed routes.

Figure 5 gives us a visual of which routes the algorithm chooses. As one can see the initial route is

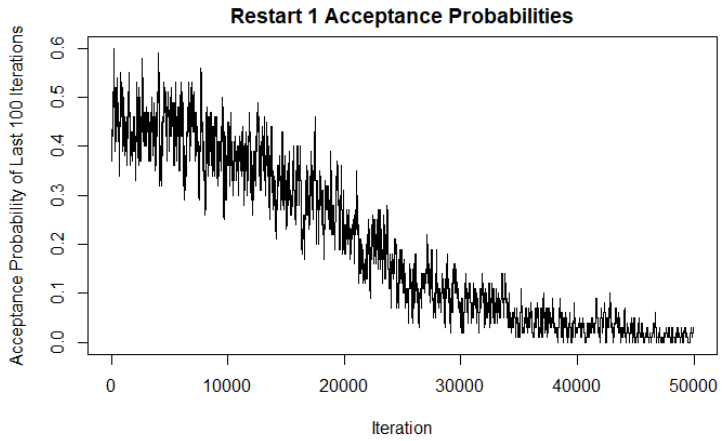


Figure 4: Rolling Acceptance Probabilities of Restart 1.

exceptionally inefficient, criss-crossing the state with a driving time three times that of the optimal route. As the algorithm iterates we see that it slowly learns to visit the two westernmost cities one after another, and also to visit the cities in and around Des Moines at the same time. However we also see the limitations of the final solution. It appears in the easternmost part of the state the final route is not ideal for three cities. This speaks to the potential limitation of generating proposals using only two cities. When the temperature is very low to achieve the optimal solution one must rearrange three cities on the route, but this is only possible through two suboptimal proposals in a row which are unlikely to be accepted.

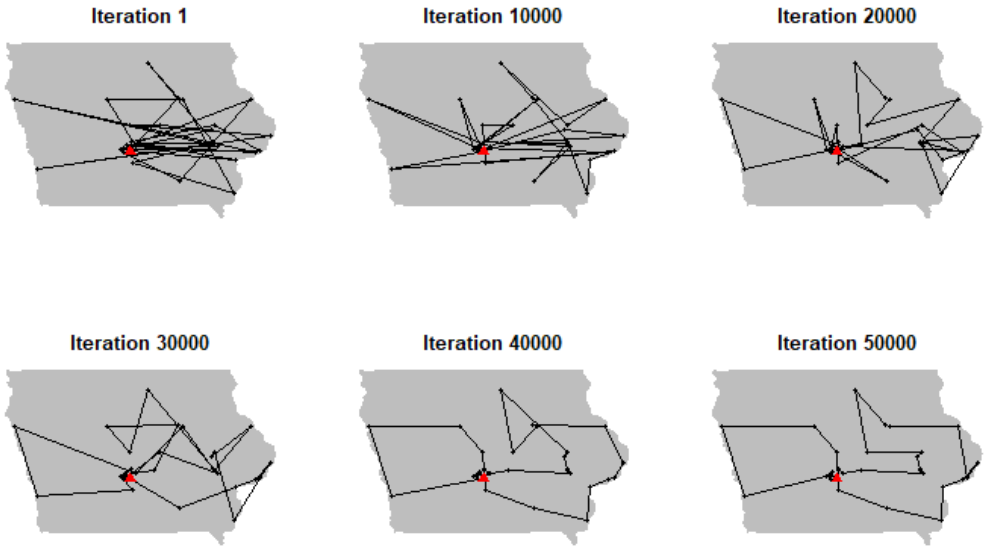


Figure 5: Restart 6 - Proposed Routes by Iteration (Start/Finish City in Red).

6.3 Higher Dimensions

As demonstrated in Section 6.1, the algorithm is able to find the optimal route for a small number of cities; however in Section 6.2 we see it comes up short in higher dimensions.

	Proposal Cities				
Total Cities	**2**	**3**	**4**	**6**	**Possible Routes**
20	19.97	20.03	19.8	21.01	2.43×10^{18}
30	24.30	22.91	23.24	23.93	2.65×10^{32}
50	36.1	35.86	35.80	42.52	3.04×10^{64}
100	71.15	66.51	75.46	94.18	9.33×10^{157}
150	109.03	106.38	117.88	144.08	5.71×10^{262}
200	154.45	149.38	167.13	210.66	7.89×10^{374}

Table 3: Minimum Driving Time vs. Total Cities & Number of Cities Involved in a Proposal

In Table 3 we see that the optimal number of cities involved in a proposal tends to be three in higher dimensions (as opposed to sometimes higher than 3 in lower dimensions). This seems to reflect a tradeoff between the fact that sometimes more than two cities are needed to give a new route that meaningfully decreases the time, and as the number of proposal cities goes up the probability of selecting the 'right' set of cities to rearrange goes down. We can see in Figure 6 that when the number of proposal cities is six, the acceptance probabilities decrease, and the algorithm spends more iterations where it accepts almost no proposals.

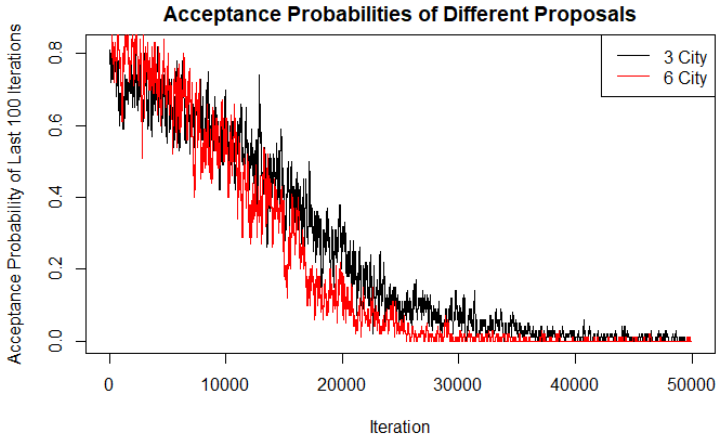


Figure 6: Acceptance Probabilities for Three vs. Six Proposal Cities (100 Cities Total).

Finally, in Figure 7 we see that the end route for one hundred cities with the number of proposal cities equal to six, looks less efficient than the case where the number of proposal cities is three.

6.3.1 The Value of Restarts

We can also see how the importance of using restarts is greater in higher dimensions than in lower dimensions. It would intuitively make sense that in lower dimensions a single restart may be sufficient because the ratio $\frac{\text{Iterations}}{\text{Number of Possible Routes}}$ is high, and a single restart may be able to explore enough of

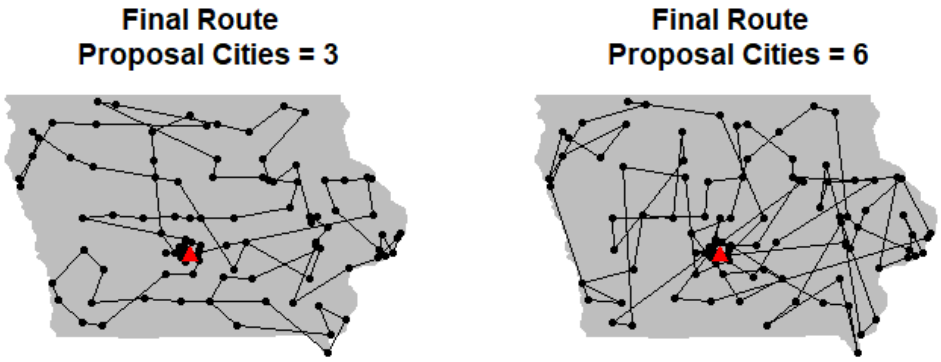


Figure 7: Final Route - 100 Cities

the possible routes to find a global, or near global, optimum. However as the dimension goes up, and the number of iterations remains constant, this ratio decreases, and it is less likely on a single run for the algorithm to find a global optimum. As we can see in Table 4 as the number of total cities goes up, the value of using restarts also goes up.

Number of Cities	**20**	**30**	**50**	**100**	**150**	**200**
Difference Between Best and Average Time	2.4%	3.8%	6.0%	4.6%	7.9%	6.0%

Table 4: Decrease in Time By Selecting the Best Time Over All Six Restarts Relative to the Average Time of All Six Restarts

7 Conclusion & Future Directions

Overall it appears MCMC methods are able to achieve optimal, or near-optimal results on the TSP in low dimensions. In higher dimensions it does a passable job, although it is clear that the solutions are provided are not optimal solutions. This is somewhat unsurprising; since MCMC is a type of random search algorithm, as the number of dimensions increases, the number of possible solutions increases very rapidly, and the chances of randomly selecting the optimal solution decreases.

One potential future avenue of analysis is attempting some sort of 'human-in-the-loop' MCMC optimization of the TSP, specifically with respect to the initial values which the algorithm takes, or adjusting final solutions. It is not too difficult for a human to string together a route that 'makes sense' (i.e. a route which does not criss-cross the state multiple times, or visits nearby cities one after another). It

is reasonable to assume that if the initial values are close to optimal, a random search algorithm like MCMC might be able to find the true optimal solution. Alternatively there may be gains to be had from humans adjusting routes at relatively low temperatures, and then continuing to run the algorithm from these modified routes.

One other avenue of analysis to look at would be to find more efficient ways to propose new routes than randomly selecting a number of cities on a route and swapping them. For example, one might alternate between selecting two cities that are far apart and swapping their positions, or selecting many cities close together, and swapping their positions. A proposal technique such as this may solve the problem where sometimes that final solution has many cities close together with a suboptimal route because under the current proposal regime, randomly selecting all of those cities at the same time is very unlikely.

References

[1] Iowa Mileage Chart. URL https://www.mileage-charts.com/chart.php?p=chart&a=NA&b=US&c=IA.

[2] Iowa Latitude and Longitude Map. URL https://www.mapsofworld.com/usa/states/iowa/lat-long.html.

[3] Marco Antonio Cruz-Chavez and Juan Frausto-Solis. Simulated Annealing with Restart to Job Shop Scheduling Problem Using Upper Bounds. In *International Conference on Artificial Intelligence and Soft Computing*, pages 860–865. Springer, 2004.

[4] Dalia Hatuqa. Why Do the Iowa Caucuses Matter?, Jan 2016. URL https://www.aljazeera.com/indepth/features/2016/01/iowa-caucuses-matter-160129112210955.html.

STA3431 Mini-Project - R Appendix

David Veitch

25/11/2019

```r
library(chron)

#### Import Driving Time Data ####
ia_times <- read.csv('IADrivingTimes.csv')

# Delete duplicate towns
# Two Oaklands (first bad)
bad_index = which(ia_times[1]=='Oakland')[1]
ia_times <- ia_times[-c(bad_index),-c(bad_index+1)]

# Two Centervilles (second bad)
bad_index = which(ia_times[1]=='Centerville')[2]
ia_times <- ia_times[-c(bad_index),-c(bad_index+1)]

# set first column as row names
ia_cities <- ia_times[,1]
ia_times <- as.matrix(ia_times[,-1])

ia_times <- as.vector(24*as.numeric(times(ia_times)))
ia_times <- matrix(ia_times,nrow=length(ia_cities),ncol=length(ia_cities))
rownames(ia_times)<-ia_cities
colnames(ia_times)<-ia_cities

# NA Values (same city to same city), fill with 0
ia_times[is.na(ia_times)]<-0

#### Import Population Data ####
ia_population <- read.csv('IAPopulation.csv',header=FALSE,row.names=1)
colnames(ia_population)<- 'Population'
rownames(ia_population)[1] <- 'Ackley'

# ia_population in order
ia_population_ordered = ia_population[order(ia_population,decreasing=TRUE),,drop=FALSE]

#### Import Latitude Longitude Data ####
ia_latlong = read.csv('IALatLong-MapsofWorld.com.csv',header=FALSE,row.names=1)
colnames(ia_latlong)<-c('lat','lon')

# Fix rownames
rownames(ia_latlong)[1] <- 'Ackley'
rownames(ia_latlong)[448]<- 'Jewell'
rownames(ia_latlong)[856]<- 'Saint Ansgar'

# merge lat/lon dataset with cities in population dataset
ia_latlong <- merge(x=ia_population,y=ia_latlong,by=0,all.x=TRUE)
rownames(ia_latlong)<-ia_latlong$Row.names
ia_latlong<-ia_latlong[,c('lat','lon')]
```

```r
# here we will output the dataframe with the top M cities
# according to population and their driving times
reduced_city_list <- function(times,population,M){
  # INPUT   - times - dataframe of driving times
  #         - population - dataframe of populations of cities
  #         - M - number of cities to use
  # OUTPUT  - dataframe of driving times for only certain cities

  # True/False vector with cities meeting minimum cutoff as True

  cities <- rownames(population[order(-population), , drop = FALSE])[1:M]
  times <- times[cities,cities]
  return(times)

}

# Example
# example_cities = reduced_city_list(ia_times,ia_population,5)
```

```r
# Evaluate the distance of a given route
route_time <- function(route,times){
  # INPUT   - times - dataframe of driving times
  #         - route - a vector of city names specifying the route one is taking
  # OUTPUT  - the total time of the route

  route_length = length(route)

  # Rearranges the time matrix with origin as rows and destination as columns
  times = times[route[1:(route_length-1)],route[2:route_length]]

  # total time of the trip, returns it
  return(sum(diag(times)))

}


# # Create a route, make it start in Des Moines
# start_route = rep(0,dim(example_cities)[1]+1)
# start_route[1] = 'Des Moines'
# start_route[length(start_route)]='Des Moines'
# start_route[2:(length(start_route)-1)] <- sample(rownames(example_cities)[rownames(example_cities)!='
#
# # Calculate Example Route Time
# route_time(start_route,example_cities)
```

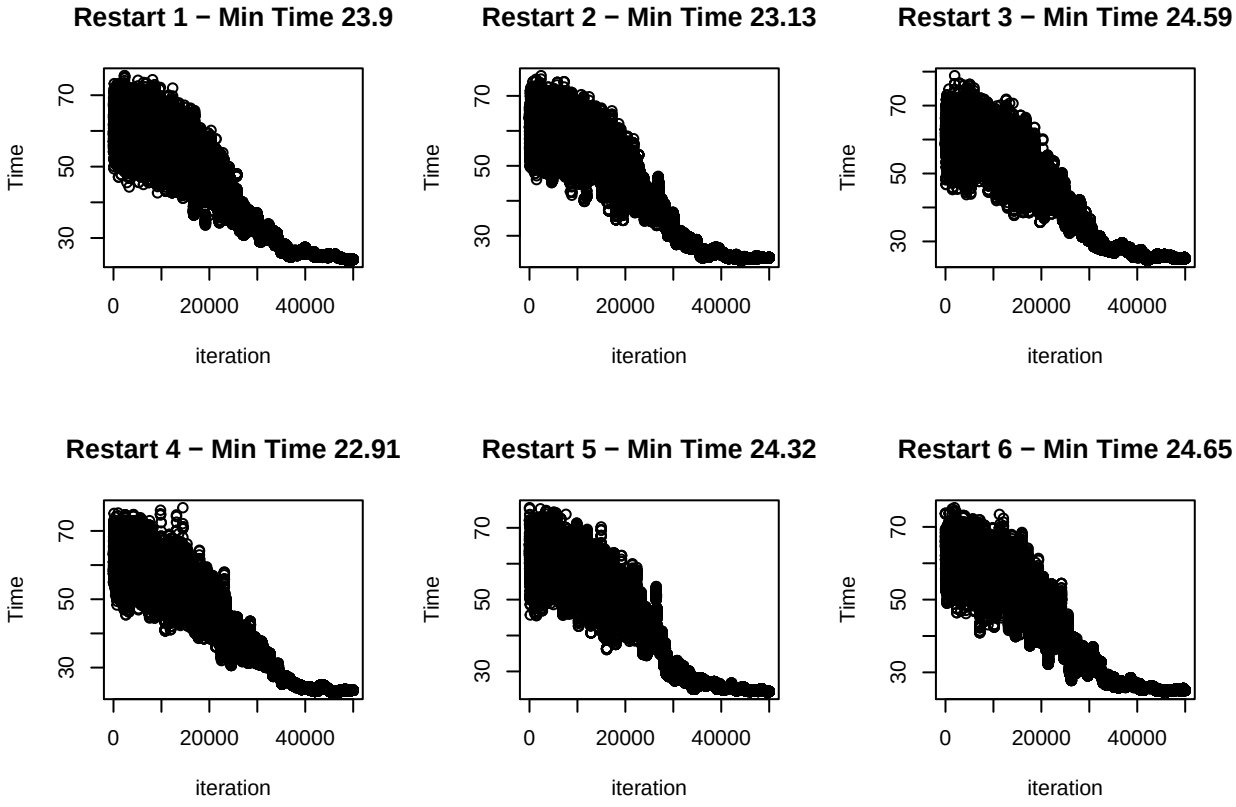
```r
generate_proposal <- function(current_route,start_constraint,end_constraint,num_swaps){
  # INPUT   - current_route - route that has currently been suggested
  #         - start_constraint - T/F if we can change the starting city
  #         - end_constraint - T/F if we can change the starting city
  #         - num_swaps - the number of indexes to swap in a given proposal
  # OUTPUT  - proposal_route - a different route with two destinations swapped

  route_length = length(current_route)
  proposal_route = current_route
```

```
  # Various cases based on if there is a start or end constraint
  if(start_constraint & end_constraint){
    # start and end constraint
    idx_swap = sample(2:(route_length-1), num_swaps, replace=FALSE)
  } else if(start_constraint){
    # start constraint
    idx_swap = sample(2:(route_length), 2, replace=FALSE)
  } else if(end_constraint){
    # end constraint
    idx_swap = sample(1:(route_length-1), 2, replace=FALSE)
  } else {
    # no constraint
    idx_swap = sample(1:(route_length), 2, replace=FALSE)
  }

  new_order = sample(idx_swap,num_swaps,replace=FALSE)
  proposal_route[idx_swap] = proposal_route[new_order]

  return(proposal_route)
}

# generate_proposal(c("Des Moines","A","B","C","D",'E','F','G',"Des Moines" ),TRUE,TRUE,3)
```

```
##########################
#### Key Parameters #####
M = 50000
num_cities = 30
cities = reduced_city_list(ia_times,ia_population,num_cities)
num_proposal_swaps = 3

# Beginning temp
start_temp=10
temp=10

# Cooling type
# GEOMETRIC, LOG, LINEAR options
cool_type = 'GEOMETRIC'
lin_temp = 0.000199999
geo_temp = .9999

# Num Restarts (1 if only one run of program)
num_restarts = 6


# Start/End city, set to NaN if none
start_city = 'Des Moines'
end_city = 'Des Moines'

############################

restart_solutions = list()
restart_route_list = list()
restart_route_time = list()
```

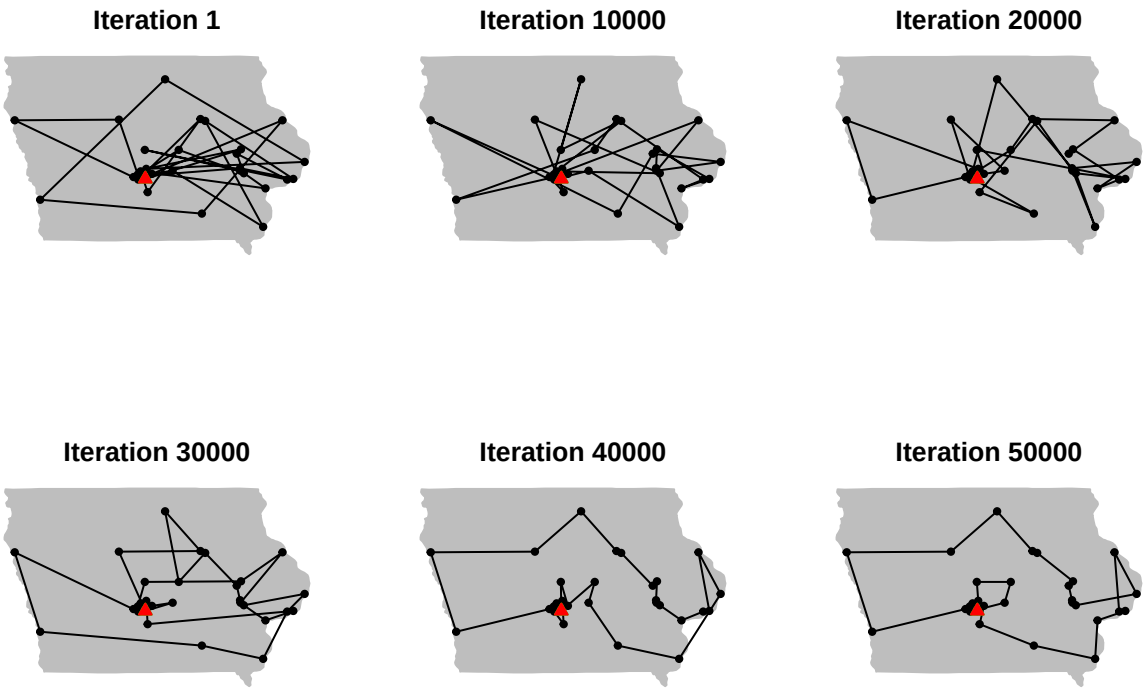
```
start_constraint = (nchar(start_city)>1)
end_constraint = (nchar(end_city)>1)

# set graphing parameters for 6 runs
par(mfrow=c(2,3))

for(restart in seq(1,num_restarts)){
  set.seed(restart)

  # list of routes and distances algorithm takes
  route_list = list()
  route_time_list = c()
  accepts=0

  # Initial Route - Have it Start in start_city, end in end city
  route = rep(0,dim(cities)[1]+1)
  route[1] = start_city
  route[length(route)]=end_city
  route[2:(length(route)-1)] <- sample(rownames(cities)[rownames(cities)!=c(start_city,end_city)],
                                       replace=FALSE)

  route_list[[1]]=route
  route_time_list[1]=route_time(route,cities)

  # the best route is currently the first one
  best_route = list(route,route_time_list[1])

  for(i in 2:M){
    # Propose a Move
    Y = generate_proposal(route,start_constraint,end_constraint,num_proposal_swaps)

     # Update temperature
    if(cool_type=='GEOMETRIC'){
      temp=start_temp*geo_temp^i
    } else if(cool_type=='LOG'){
      temp = start_temp/log(1+i)
    } else if(cool_type=='LINEAR'){
      temp = temp-lin_temp
    }

    # Accept/reject based on inverse distance
    U = runif(1)
    A = ((exp(-1*route_time(Y,cities)))/(exp(-1*route_time(route,cities))))^(1/temp)

    if(U<A){
      route = Y
      accepts = accepts+1
    }

    route_list[[i]] = route
    route_time_list[i] = route_time(route,cities)

  }
```

```
  plot(route_time_list,main=paste('Restart',restart,'-','Min Time',
                                  round(min(route_time_list),2)),
       xlab='iteration',ylab='Time')

  restart_solutions[[restart]] = route_list[[which.min(route_time_list)[1]]]
  restart_route_list[[restart]] = route_list
  restart_route_time[[restart]] = route_time_list
}
```

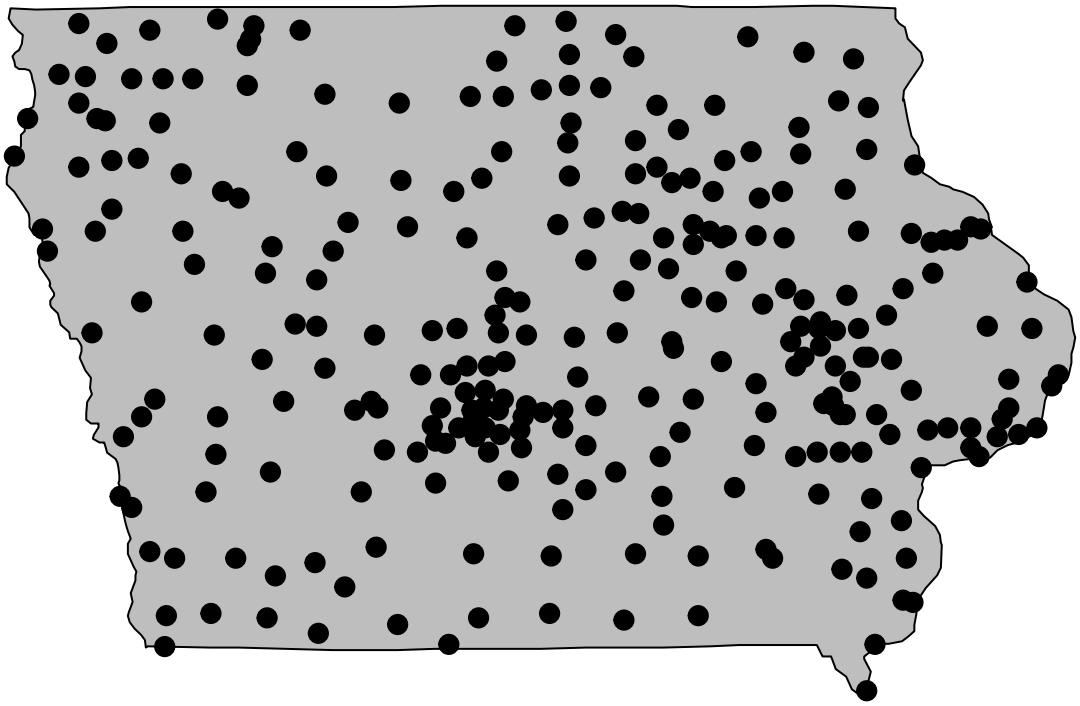



```
library(maps)
library(mapdata)
library(mapproj)

par(mfrow=c(2,3))

# Which iteration of the algorithm to run
restart_number = 4
route_to_map = restart_route_list[[restart_number]]

### Map Progression ####
cities_to_map = ia_latlong[route_to_map[[1]],]

for(i in c(1,10000,20000,30000,40000,50000)){
  cities_to_map = ia_latlong[route_to_map[[i]],]
  par(mar=c(0,0,0,0))
  map("state","Iowa",fill=TRUE,col='grey',border='grey')
```

```
  points(x=cities_to_map[,c('lon')],y=cities_to_map[,c('lat')],
       col="black", cex=1, pch=20)
  lines(x=cities_to_map[,c('lon')],y=cities_to_map[,c('lat')], col = "black")

  title( main = paste('Iteration',i),line=1)

  # plot start/finish
  points(x=cities_to_map[,c('lon')][1],y=cities_to_map[,c('lat')][1],
       col="green", cex=1, pch=24,bg='green')
  points(x=cities_to_map[,c('lon')][length(cities_to_map[,1])],
         y=cities_to_map[,c('lat')][length(cities_to_map[,1])],
       col="red", cex=1, pch=24,bg='red')
}
```



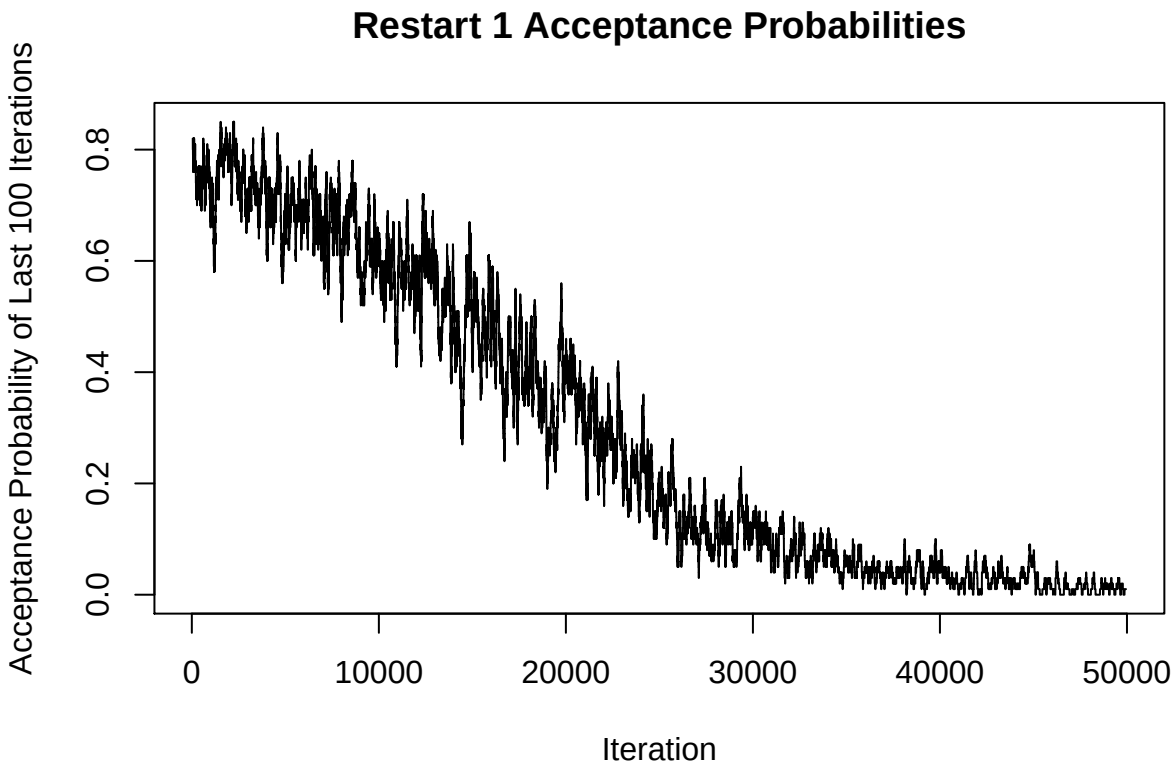

```
# iowa cities
par(mar=c(0,0,0,0))
map("state","Iowa",fill=TRUE,col='grey')
points(x=ia_latlong[,c('lon')],y=ia_latlong[,c('lat')],
       col="black", cex=2, pch=20)
```



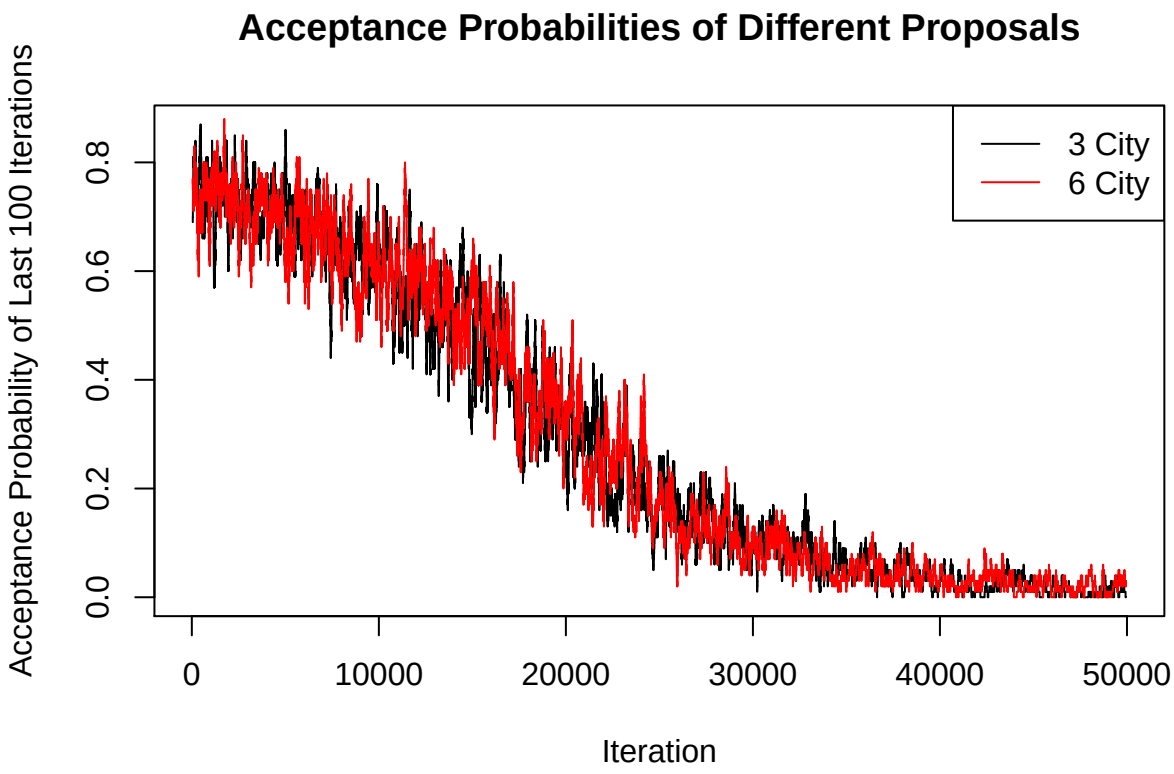
```
# Code to find the optimal route for 10 cities

# library(combinat)
#
# route_list = list()
# route_time_list = c()
#
# route = rep(0,dim(cities)[1]+1)
# route[1] = start_city
# route[length(route)]=end_city
#
# route[2:(length(route)-1)] <- sample(rownames(cities)[rownames(cities)!='Des Moines'],replace=FALSE)
#
# route_list[[1]]=route
# route_time_list[1]=route_time(route,cities)
#
# perm_routes = permn(route[2:11])
#
# for(i in seq(1,length(perm_routes))){
#   perm_routes[[i]] = c('Des Moines',perm_routes[[i]],'Des Moines')
# }
#
# route_times = c()
#
# for(i in seq(1,length(perm_routes))){
#   route_times[i] = route_time(perm_routes[[i]],cities)
```

```
# }
```

```
accepts = (diff(restart_route_time[[1]])!=0)

restart_plot = plot(filter(accepts, rep(1/100,100)),
                    main='Restart 1 Acceptance Probabilities',
                    xlab='Iteration',
                    ylab='Acceptance Probability of Last 100 Iterations')
```




```
accepts_3prop = (diff(restart_route_time[[4]])!=0)
accepts_6prop = (diff(restart_route_time[[6]])!=0)

plot(filter(accepts_3prop, rep(1/100,100)),
                    main='Acceptance Probabilities of Different Proposals',
                    xlab='Iteration',
                    ylab='Acceptance Probability of Last 100 Iterations')
lines(filter(accepts_6prop, rep(1/100,100)),col='red')
legend('topright', legend=c("3 City", "6 City"),
       col=c("black", "red"), lty=1, cex=1)
```


Acceptance Probabilities of Different Proposals
Acceptance Probability of Last 100 Iterations
0.0
0.2
0.4
0.6
0.8
0
10000
20000
30000
40000
50000
Iteration
3 City
6 City