Some Simulated Annealing Applications And Card Shuffling

Reihaneh Entezari

April 2012

Department of Statistics - University of Toronto
Supervisor: Prof J.Rosenthal

Contents

1 Introduction

Nowadays, MCMC is one of the most commonly used methods that has grasped many statisticians interest with its extensive applications. Specifically, Simulated Annealing(SA) is a well known algorithm used for optimization problems that has various applications such as sudoku puzzles and the travelling salesman problem which will be done in this paper. Card shuffling is another interesting application of MCMC, which will also be discussed here.

1.1 Simulated Annealing

As mentioned above, simulated annealing is an optimization method, where its main goal is to find (or approximate) the global optimal solution to the problem. In SA, the problem of interest is usually in a form of a function called "Cost" where we wish to either find its global minimum or maximum. SA starts with an initial guess for the solution and attempts to replace the current solution by another random solution (the step where we "Propose"). We accept the proposed solution if the cost function moves in the direction closer to minimizing (or maximizing) it. This is done by accepting with a probability that depends on both, the differences in the cost function and also a parameter T (called temperature) that decreases (cools down) after each iteration so that at high temperature it can search more for a solution and as the temperature decreases, it searches around the solution to find the actual optimal point. There are different cooling schedules, where the common ones are linear cooling and geometric cooling. In this paper we will use geometric cooling, where there is an initial value for T, and that after each iteration it is multiplied by a constant. Here we will see how SA is implemented in R to solve problems like the Travelling Salesman Problem (TSP) and Sudoku Puzzles.

1.2 Card Shuffling

Card Shuffling has become one of the most engaging problems in many games involved with a deck of cards. The goal is to shuffle the cards as many times as needed so that after several shuffles, no one can know the order of the cards. In mathematical words, any permutation of the cards are likely to appear at final, with the same probability (having Uniform distribution).

In this paper, we will introduce different ways of shuffling a deck of cards and will implement them in R to see that they will converge to the uniform distribution after some amount of iterations.

2 Card Shuffling

Scrambling the order of a deck of n cards in any manner can be called a *shuffle*. In card shuffling, the state space Ω is the set of all possible permutations of n cards, S_n . The main goal is to choose a specific way of shuffling so that after several shuffles the chain will converge to the uniform distribution π (P(any order) $=\frac{1}{|S_n|}=\frac{1}{n!}$). There are some popular ways of shuffling a deck of cards, described as below :

- **Random Transposition Shuffle:** For each shuffle, we choose two cards i and j uniformly at random with replacement and swap their places. In this case the Markov Chain will be irreducible (since any permutation is a product of transpositions) and aperiodic (since it's possible that i=j). Also swapping the same cards twice will give the first permutation so *P(x, y)=P(y, x)* for every pair of x, y and thus P is symmetric, hence reversible w.r.t. the uniform distribution. So it's stationary distribution is uniform and thus from theorem, the chain will converge to the uniform distribution.

- **Top-to-Random Shuffle:** In each iteration we take the top card and insert it at one of the n positions of the deck of cards, uniformly at random. Again the Markov Chain is irreducible and aperiodic, but here P is not symmetric anymore since if the top card is inserted somewhere in between, we cannot get back to the same state in one step. But starting at any state x, the chain can move to one of n states in one step, since the top card has n positions to go to, each with probability $\frac{1}{n}$. Similarly, for every state y, it comes back from exactly one of the n states each with probability $\frac{1}{n}$. Thus every row and column of the transition probability matrix P sums to 1. This matrix is called *doubly stochastic* and so from a lemma, the Markov Chain is reversible w.r.t. the uniform distribution, hence the chain will converge to the uniform distribution.

- **Riffle Shuffle: GSR-Model** In each iteration we (1) split the deck of cards into two sets according to the binomial distribution *Bin(n,1/2)*, one in the left hand L and one in the right hand R. (2) Drop the cards in a sequence where the next card comes from the left hand with probability $\frac{|L|}{|L|+|R|}$ where $\mid L \mid$ and $\mid R \mid$ are the number of cards remaining in the left and right hand, respectively at that step of dropping the cards. It has been proved that the second step is equivalent to selecting an interleaving of the two parts uniformly at random preserving the order of the cards in each hand.

These three different ways of shuffling a deck of cards, has been implemented in R and can be seen in appendix 1. Some results of the printout are discussed below:

2.1 Results

As it is seen in Appendix 1, there are three different functions, each implementing one of the three shuffling algorithms explained above. Each function has been run for different number of deck of cards (n) and different amount of iterations (M). From the printout, it is clear as M increases, the chain converges to the uniform distribution π, just as we had expected from theorems. So mainly, after the needed amount of shuffles, any order of the deck of cards are likely to appear with the same probability (Uniform Distribution). The program output can be seen in Appendix 1.

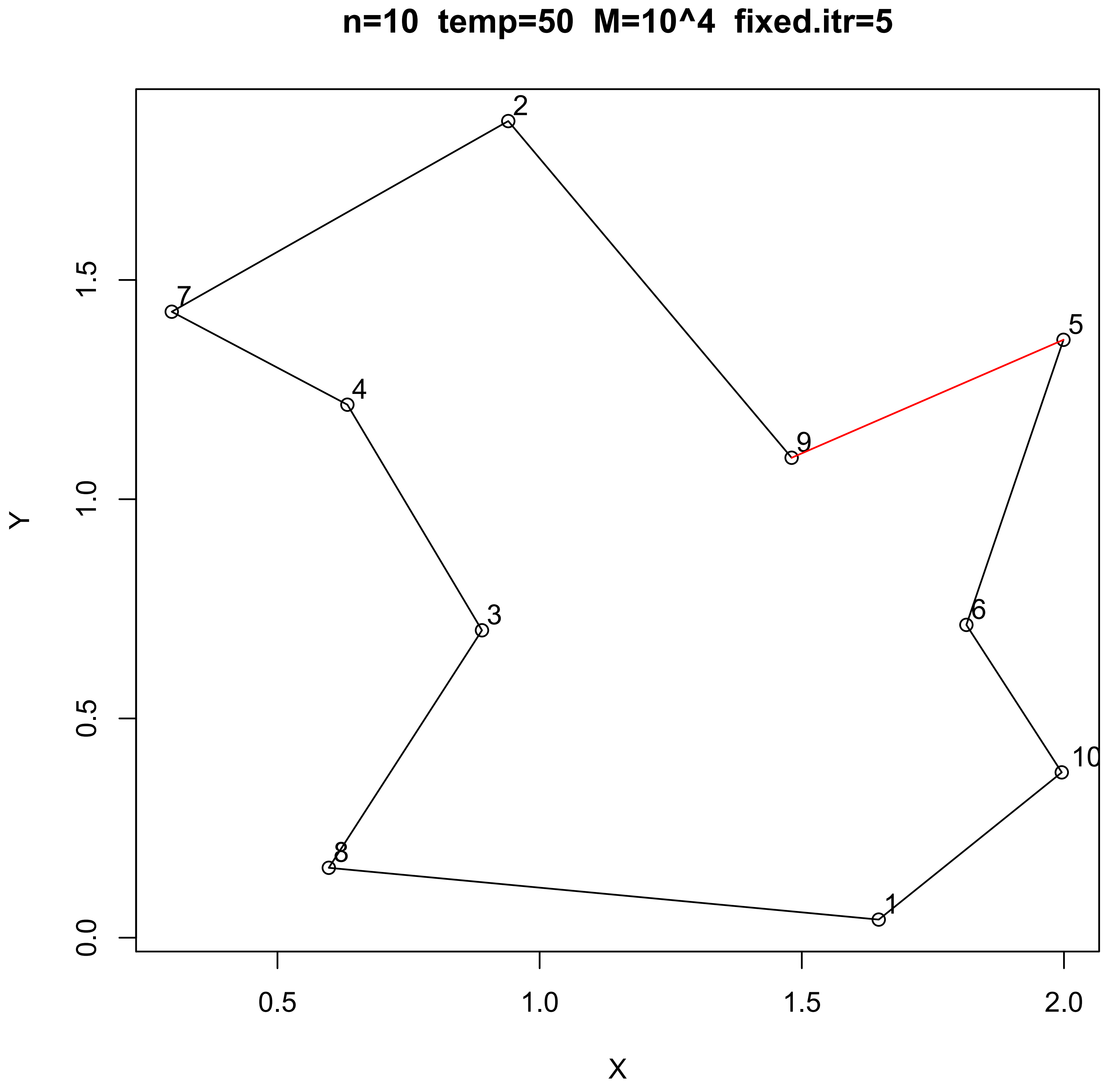
3 Travelling Salesman Problem (TSP)

Travelling Salesman Problem (TSP) is one the most popular optimization problems which can be solved by simulated annealing method. Suppose that there are n cities in a region of a country, and that we have the pairwise distances between the cities. The goal is to find the smallest path starting from one city, visiting all the other cities exactly once and finally returning to the origin city. This problem also has several applications such as planning, DNA sequencing... In this problem, the cost function is the travelling distance for a specific path and the possible paths can be seen as a permutation over the n cities. Starting with an initial order of the cities, we propose by randomly choosing two cities and swapping their positions. Hence the new order will be accepted with probability min(1, $\exp^{-dC/T}$). The cooling schedule used here is geometric cooling.This has been implemented in R using the

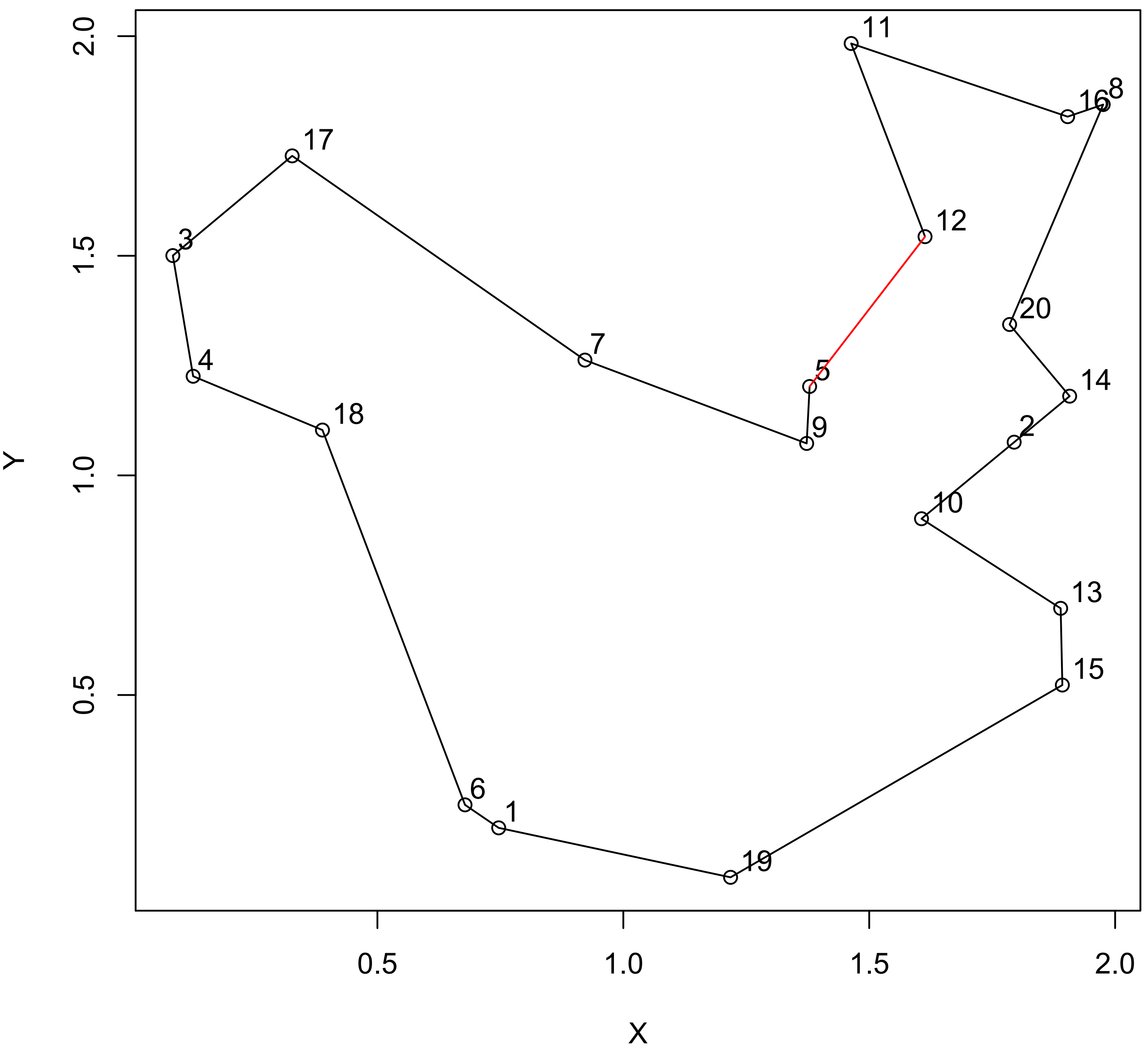
Euclidean distance space and can be seen in appendix 2. Some results of using simulated annealing for this problem are discussed below:

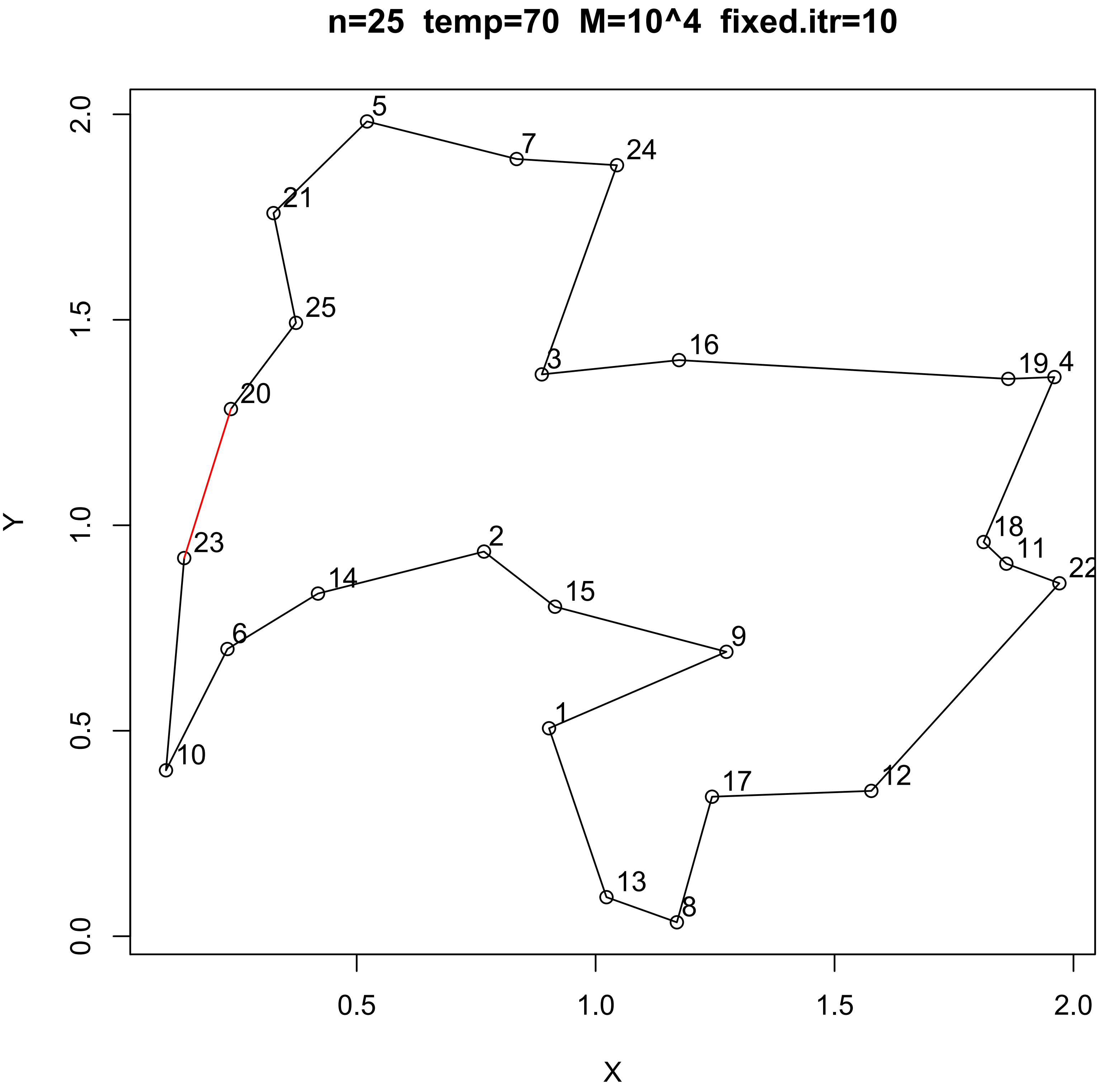
3.1 Results

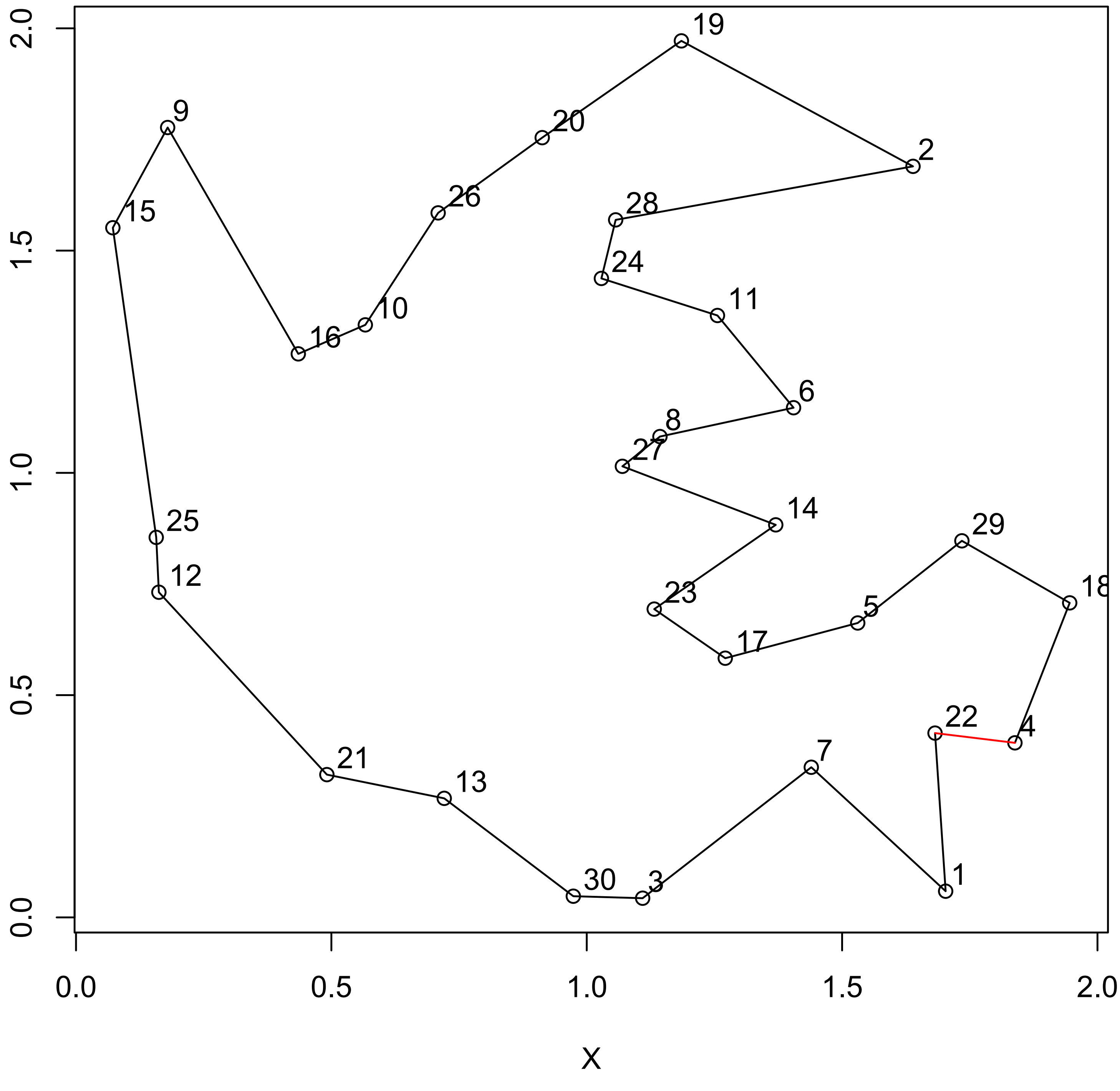
As seen in appendix 2, TSP has been implemented in R with simulated annealing. The code works well for n up to 25 cities, but as the number of cities increases, it reaches to a solution close to the actual solution, and not the true optimal path. But for it to not get stuck in a local minimum, it is better to do SA for a fixed temperature for some number of iterations and then decrease the temperature. Here iterations 5,10,20 and 30 are tested to do SA for a fixed temperature before decreasing the temperature. This has worked out better than before, since it gave the optimal path for $n = 30$ and almost for $n = 40$ (except two cities should have been changed). To get to the optimal path for larger n, we need to do more iterations for fixed temperatures and since "for" loops are much slow in R, it will take a while to run the program. Some parameters have been changed and tested such as the number of total iterations (M=10^4,10^6) and also the initial temperature (temp=100,80,70,50 which 70 worked out better most of the time). The acceptance rates has also been computed in the code, where they are usually around 0.4. Below are some graphs showing the optimal path with different number of cities and starting temperatures (the red line indicates the end of the path). The last graph is for 40 cities and seems very close to the optimal (2 cities might have to be changed). Also some graphs are showing the cost function (for ten cities) over all iterations and how it decreases to zero. Finally the last graph is the graph of the temperature showing how it cooled down :


n=10 temp=50 M=10^4 fixed.itr=5
Y
X
0.0
0.5
1.0
1.5
2.0
1
2
3
4
5
6
7
8
9
10

n=20 temp=50 M=10^4 fixed.itr=10

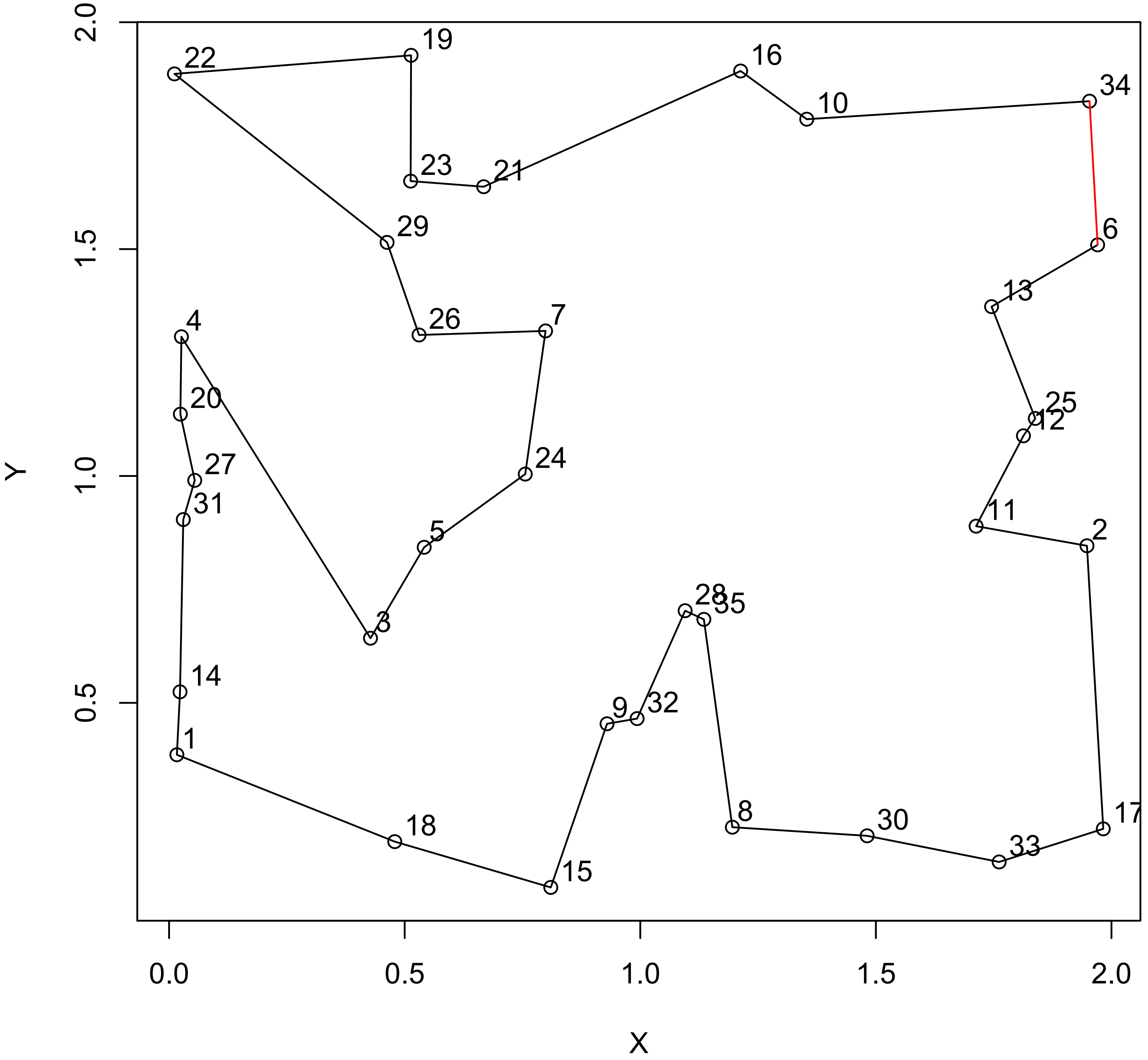


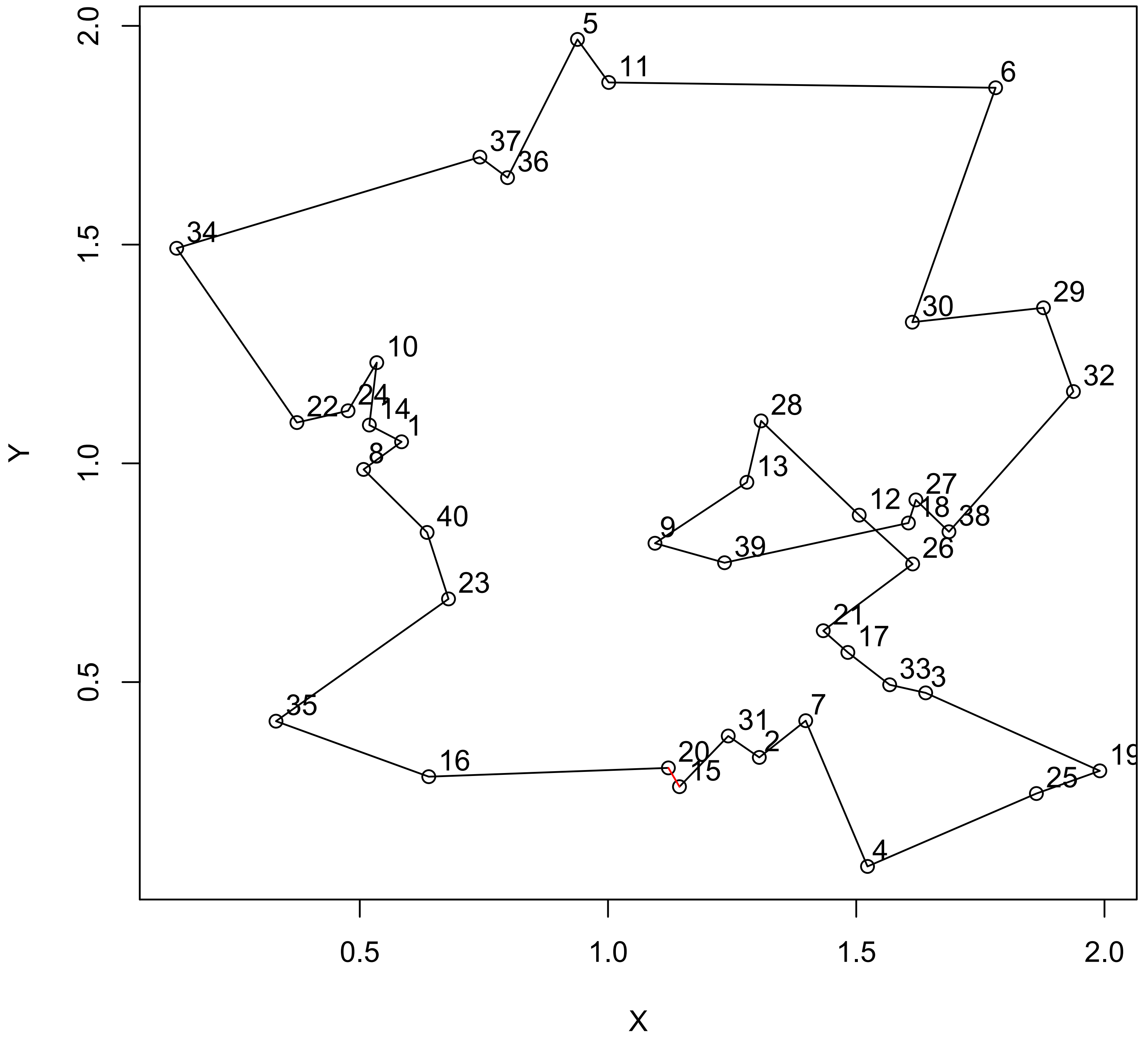

n=25 temp=70 M=10^4 fixed.itr=10
X
Y
0.5
1.0
1.5
2.0
0.0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25



19
20
2
9
26
28
15
24
11
10
16
6
8
27
14
29
25
23
18
5
12
17
22
4
21
7
13
30
3
1
0.0
0.5
1.0
1.5
2.0
X
Y

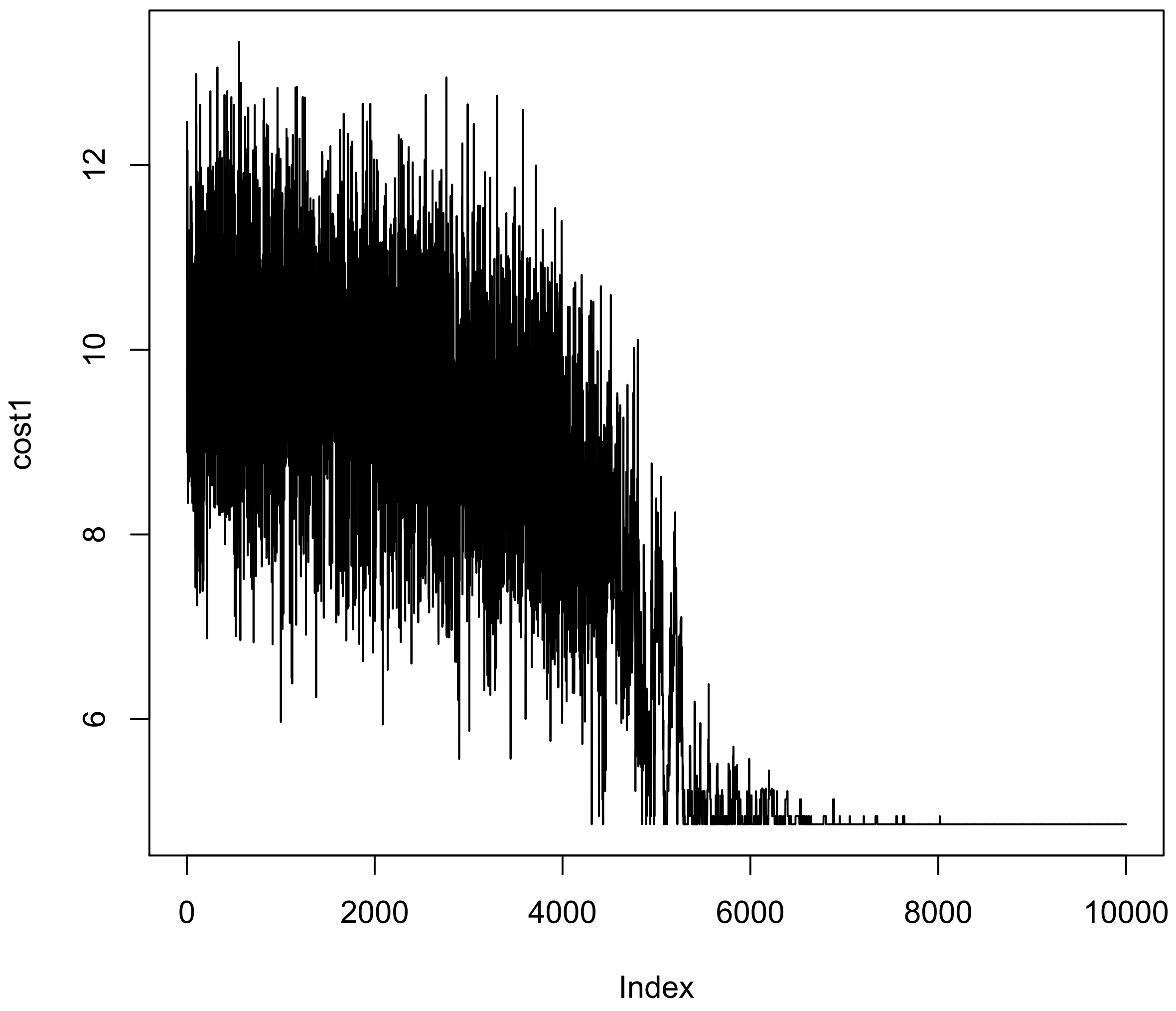
n=35 temp=50 M=10^4

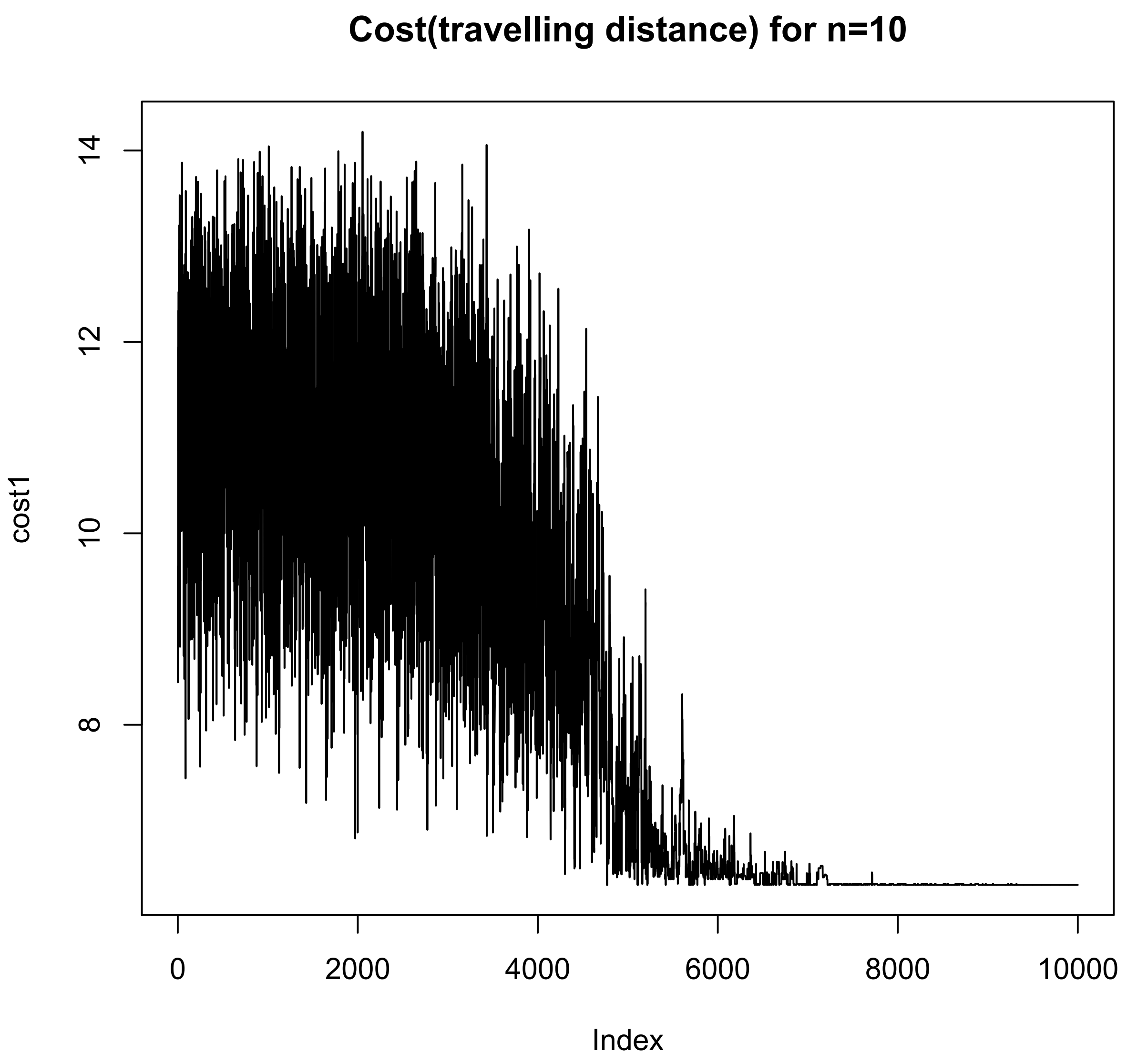


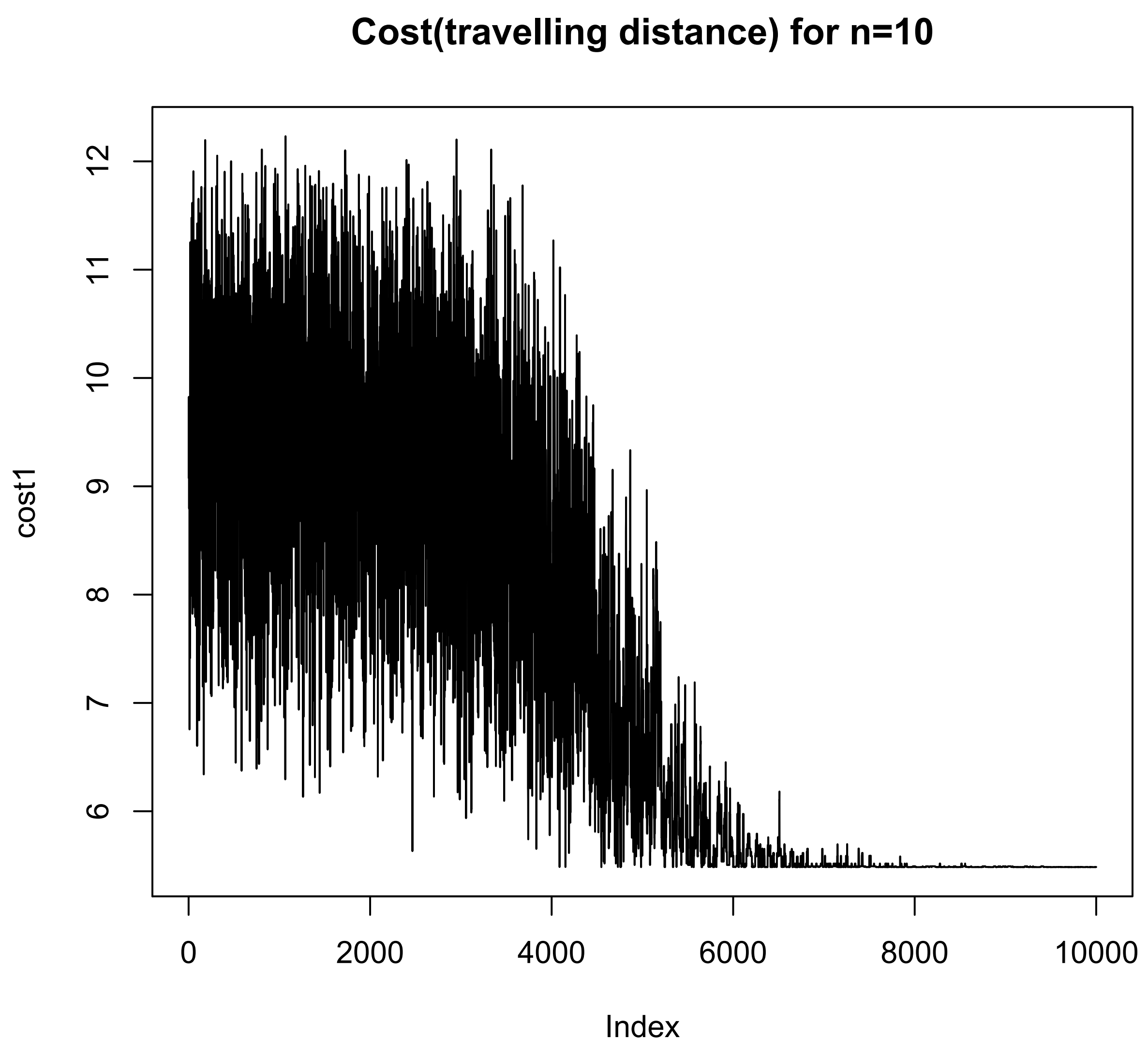


Y
X
0.5
1.0
1.5
2.0
0.5
1.0
1.5
2.0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

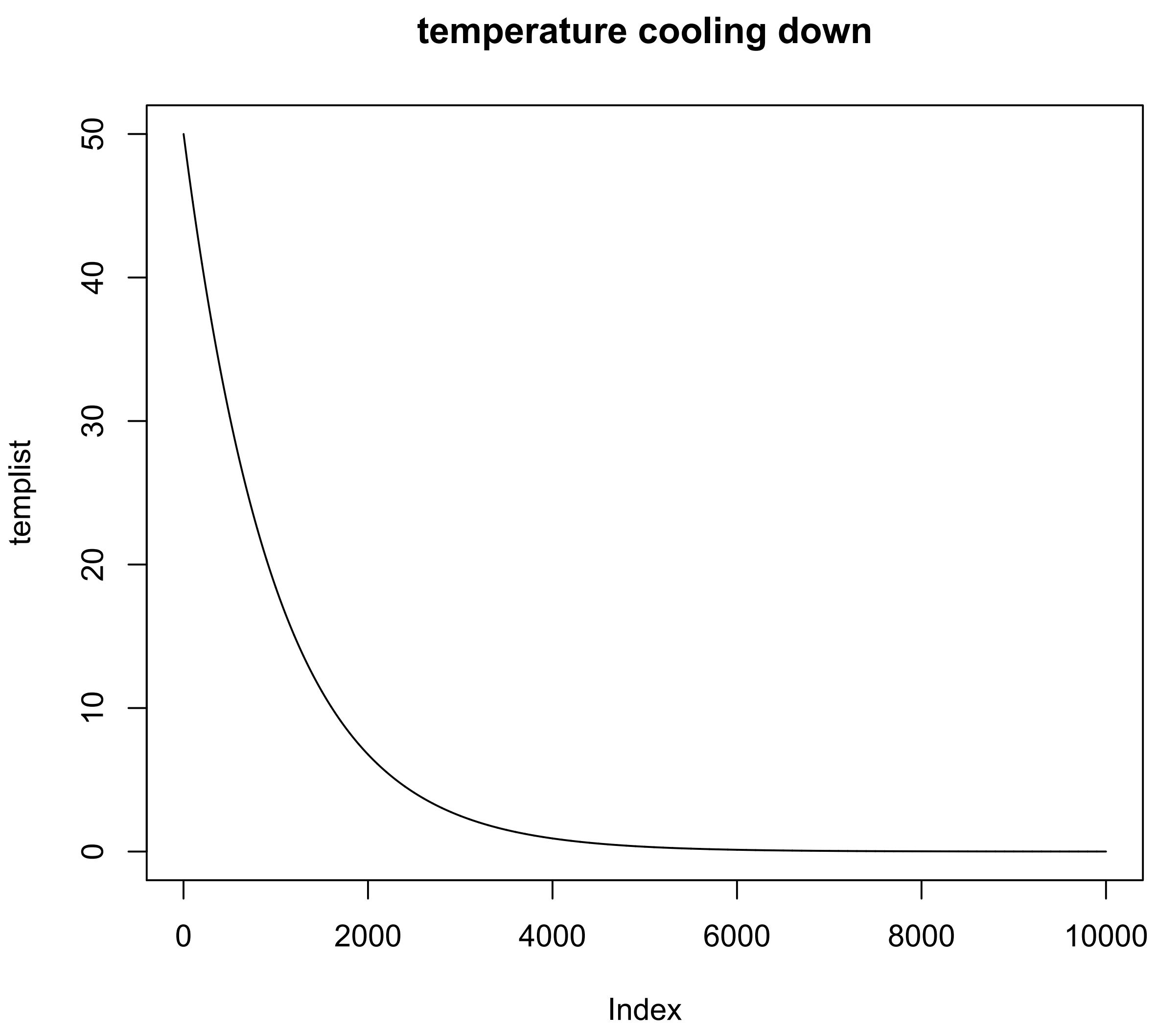
Cost(travelling distance) for n=10




Cost(travelling distance) for n=10
cost1
Index
8
10
12
14
0
2000
4000
6000
8000
10000



Cost(travelling distance) for n=10
cost1
Index
0
2000
4000
6000
8000
10000
6
7
8
9
10
11
12

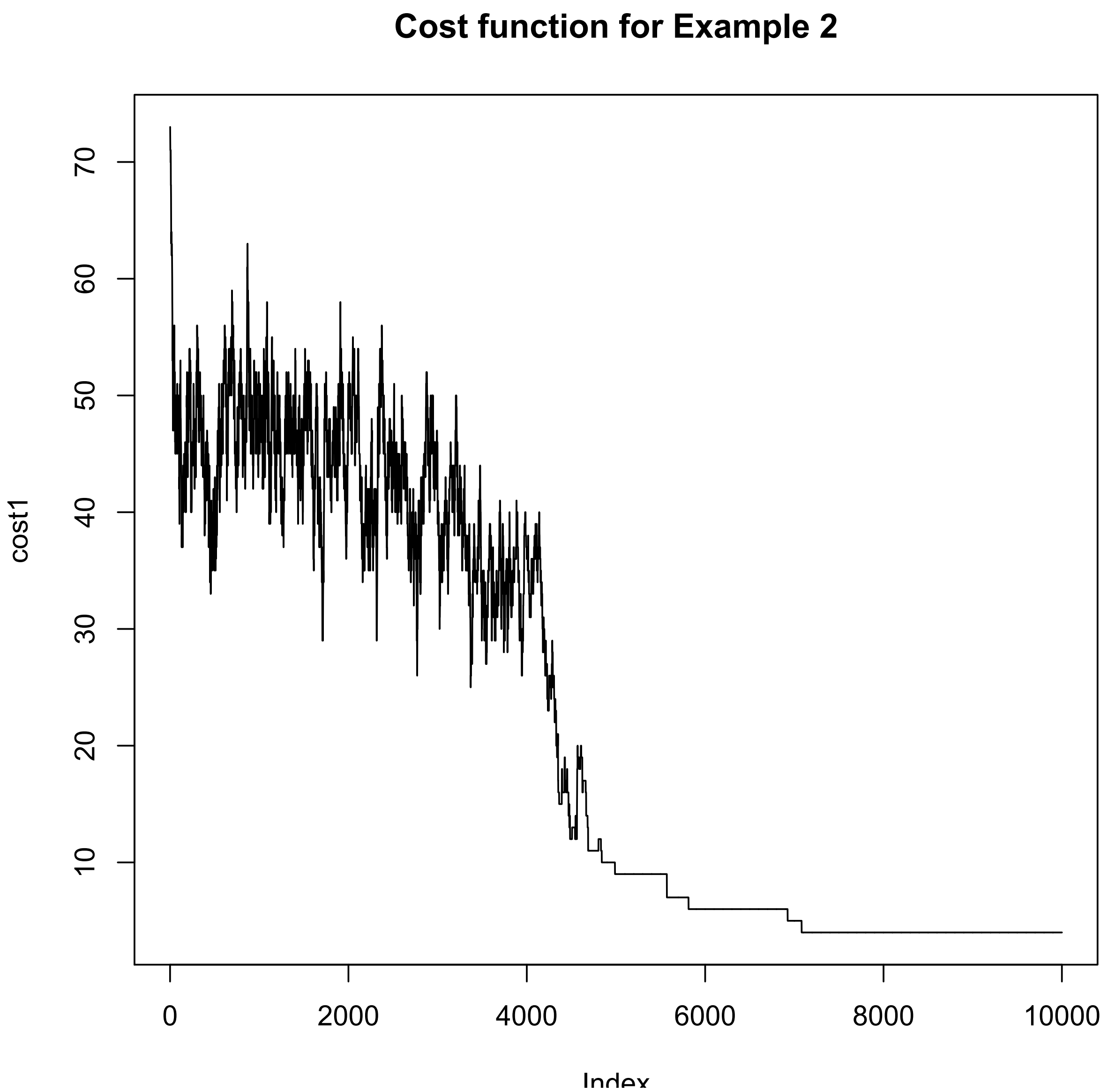

temperature cooling down
templist
0
10
20
30
40
50
0
2000
4000
6000
8000
10000
Index

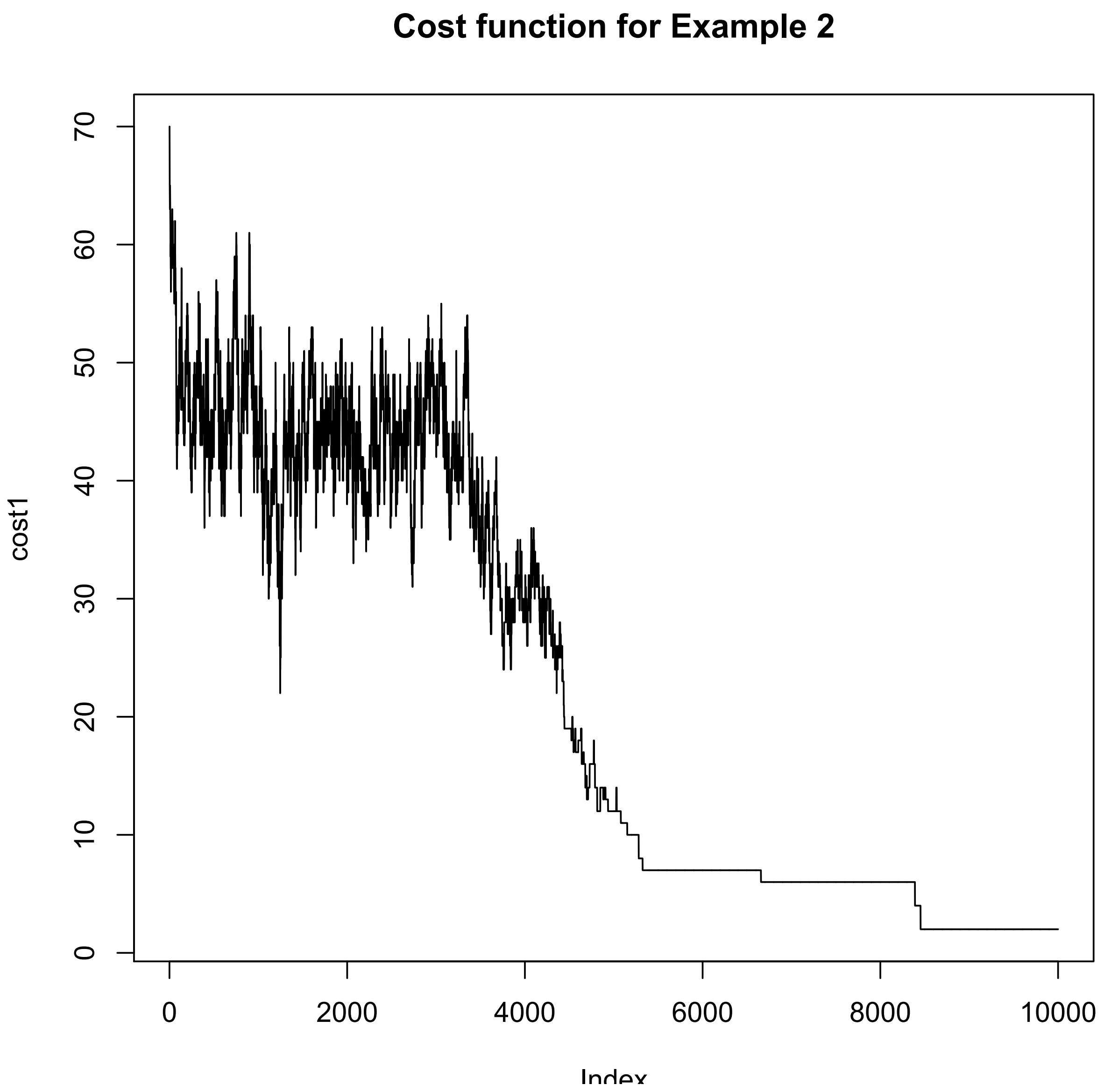
4 Sudoku Puzzles

Solving Sudoku Puzzles is another engaging problem these days. They range in difficulty from easy to very challenging; the hardest puzzles have the most empty cells. A sudoku puzzle is a 9×9 grid which is filled with digits 1,...,9 such that each digit is seen exactly once in each row, column, and a 3×3 block. Here we have used simulated annealing to reach a solution for the puzzle. The state space is a full 9×9 grid, and the cost function is the total number of digits repeated more than once in each row, column,and block. So we want the cost function to reach its minimum zero. To propose, we have to choose two cells and swap their contents, but here it's important that which two cells are chosen to swap since some cells are more trouble making cells. According to the paper " Techniques for solving sudoku puzzles" by Eric C Chi and Kenneth Lange, to ensure that the most trouble making cells are more likely to be chosen to swap, they can be chosen non-uniformly with probability proportional to $\exp(i)$, where i is the number of constraints on a cell, which is the number of times the content of that cell has been repeated in its row, column and block. We start with an initial guess for the puzzle (which also has to satisfy that each digit has exactly 9 copies of itself in the overall 9×9 grid) with respect to some clue puzzle and at each step n, we swap to propose with a new grid B that has cost c(B). The proposal is accepted with probability $min(1, \exp((c(B_n) - c(B))/T_n))$. Although there are websites that have programmed the sudoku puzzle in a similar way that it has been done here (since here it has been followed from the paper mentioned in the reference category), but I have implemented it in R too, and can be seen in appendix 3. Some results are discussed as below.

4.1 Results

As seen in Appendix 3, solving sudoku puzzles has been implemented in R using simulated annealing. However, it usually reaches a solution very close to the actual solution. The same situation as for the TSP, happens here, so for SA not to get stuck in a local minimum, it is better to iterate for a fixed temperature before decreasing the temperature. Again since "for" loops are slow in R, this can't be done fast and it takes a long time for the program to run. However, in the appendix, this code has been ran for two different sudoku puzzles (two different clue matrices), which has actually solved the first one, but the second one has remained with cost equal to 2 (which is very close to the solution and needs more runs for fixed temperature). In most of the time we are left with a small amount of cost. Some parameters has been changed and tested, for example the initial temperature, the number of iterations. The acceptance rates has also been computed and they are around 0.37 for the second example. Below are graphs showing the cost function for the second example (clue matrix) which is decreasing to zero :


Cost function for Example 2
cost1
Index
0
2000
4000
6000
8000
10000
10
20
30
40
50
60
70


Cost function for Example 2
cost1
Index
0
10
20
30
40
50
60
70
0
2000
4000
6000
8000
10000

5 References

1. ” Techniques for solving Sudoku Puzzles” by ERIC C. CHI and KENNETH LANGE

2. http://www.cs.berkeley.edu/ sinclair/cs294/n2.pdf

3. http://people.csail.mit.edu/costis/6896sp11/lec2s.pdf

4. wikipedia

6 Appendix

6.1 Appendix 1

The code starts on next page

```
> #### Random Transposition
>
> run<-function(n,M){
+ a=seq(1,n) # initial value
+ d=seq(1,n) #a fixed state of the chain to keep track of the # of times it appears
+ count=0
+ for(i in 1:M){
+     if( all(a==d) ) count<-count+1 # counting the # of times d appears
+     b=sample(1:n,2,replace=T) # choosing 2 cards uniformly at random with replace
+     result=a
+     result[b[1]]<-a[b[2]] # swapping
+     result[b[2]]<-a[b[1]] # swapping
+     a<-result
+
+ }
+ cat("fraction=",count/M,"\n") # the fraction of times d has appeared in M iterations
+ cat("actual pi=",1/factorial(n)) # the actual pi that will converge to
+ }
> run(4,10^3)
fraction= 0.028
actual pi= 0.04166667
> run(4,10^4)
fraction= 0.0435
actual pi= 0.04166667
> run(4,10^5)
fraction= 0.04135
actual pi= 0.04166667
> run(4,10^6)
fraction= 0.041808
actual pi= 0.04166667
> run(5,10^3)
fraction= 0.009
actual pi= 0.008333333
> run(5,10^4)
fraction= 0.0094
actual pi= 0.008333333
> run(5,10^5)
fraction= 0.00878
actual pi= 0.008333333
> run(5,10^6)
fraction= 0.008381
actual pi= 0.008333333
> run(7,10^3)
fraction= 0.001
actual pi= 0.0001984127
> run(7,10^4)
fraction= 2e-04
```

```
actual pi= 0.0001984127
> run(7,10^5)
fraction= 0.00025
actual pi= 0.0001984127
> run(7,10^6)
fraction= 0.000188
actual pi= 0.0001984127
> run(10,10^6)
fraction= 1e-06
actual pi= 2.755732e-07
> run(10,10^7)
fraction= 2e-07
actual pi= 2.755732e-07
>
```

```
> ### Top-to-Random
>
> run1<-function(n,M){
+ a=seq(1,n) # initial value
+ d=seq(1,n) #a fixed state of the chain to keep track of the # of times it appears
+ count=0
+ for(i in 1:M){
+     if( all(a==d) ) count<-count+1  # counting the # of times d appears
+ b=sample(1:n,1) #choosing a position from the n positions to replace the top card
+     result=a
+     result[b[1]]<-a[1] # replacing the top card to the random place
+     if(b[1]>1){ # this if statement happens when the state will have to change
+     for(j in 1:(b[1]-1)){
+           result[j]<-a[j+1] # shifting the cards before the random place one to the left
and the rest remain
+     }
+     }
+     a<-result
+
+ }
+ cat("fraction=",count/M,"\n") # the fraction of times d has appeared in M iterations
+ cat("actual pi=",1/factorial(n)) # the actual pi that will converge to
+ }
> run1(4,10^3)
fraction= 0.038
actual pi= 0.04166667
> run1(4,10^4)
fraction= 0.0408
actual pi= 0.04166667
> run1(4,10^5)
fraction= 0.0425
actual pi= 0.04166667
> run1(4,10^6)
fraction= 0.041414
actual pi= 0.04166667
> run1(5,10^3)
fraction= 0.01
actual pi= 0.008333333
> run1(5,10^4)
fraction= 0.0077
actual pi= 0.008333333
> run1(5,10^5)
fraction= 0.0089
actual pi= 0.008333333
> run1(5,10^6)
fraction= 0.008191
actual pi= 0.008333333
```

```
> run1(7,10^3)
fraction= 0.001
actual pi= 0.0001984127
> run1(7,10^4)
fraction= 5e-04
actual pi= 0.0001984127
> run1(7,10^5)
fraction= 0.00024
actual pi= 0.0001984127
> run1(7,10^6)
fraction= 0.000184
actual pi= 0.0001984127
> run1(10,10^6)
fraction= 1e-06
actual pi= 2.755732e-07
> run1(10,10^7)
fraction= 2e-07
actual pi= 2.755732e-07
```

```
> ### Riffle Shuffle
>
> run2<-function(n,M){
+ a=seq(1,n) # initial value
+ count=0
+ result=rep(0,n)
+ b=seq(1,n) #a fixed state of the chain to keep track of the # of times it appears
+ library(gtools)
+ for(i in 1:M){
+  if( all(a==b) ) count<-count+1 # counting the # of times b appears
+     j=rbinom(1,n,0.5) # choosing where to split the deck into two parts
+      if(j>0 && j<n){ # the state will change
+      a1=a[1:j]  # cards in the left hand L
+      a2=a[(j+1):n] # cards in the right hand R
+      A=combinations(n,j) # choosing j positions from n positions to replace with L
+      r=sample(1:(nrow(A)),1) # picking at random which j positions
+      t=A[r,]
+      result[t]=a1 # replacing L to the j positions chosen
+      result[-t]=a2 # replacing R to the remaining
+ #    t=sort(sample(1:n,j)) another way to choose j positions
+ #     result[t]=a1
+ #      result[-t]=a2
+      a<-result
+      }
+
+ }
+ cat("fraction=",count/M,"\n") # the fraction of times d has appeared in M iterations
+ cat("actual pi=",1/factorial(n)) # the actual pi that will converge to
+
+ }
> run2(4,10^3)
fraction= 0.046
actual pi= 0.04166667
> run2(4,10^4)
fraction= 0.044
actual pi= 0.04166667
> run2(4,10^5)
fraction= 0.04124
actual pi= 0.04166667
> run2(4,10^6)
fraction= 0.041407
actual pi= 0.04166667
> run2(5,10^3)
fraction= 0.01
actual pi= 0.008333333
> run2(5,10^4)
fraction= 0.0074
```

```
actual pi= 0.008333333
> run2(5,10^5)
fraction= 0.00876
actual pi= 0.008333333
> run2(5,10^6)
fraction= 0.008395
actual pi= 0.008333333
> run2(7,10^3)
fraction= 0.001
actual pi= 0.0001984127
> run2(7,10^4)
fraction= 3e-04
actual pi= 0.0001984127
> run2(7,10^5)
fraction= 0.00012
actual pi= 0.0001984127
> run2(7,10^6)
fraction= 0.000188
actual pi= 0.0001984127
```

6.2 Appendix 2

The code starts on next page

```
> ### Travelling Salesman Problem
>
> n=10 # number of cities
> M=10^4 # number of iterations
> temp=50 # initial temperature
> finaltemp=0.1
> tempfactor= (temp/finaltemp)^(-1/M)
> fixed.itr=5 # number of iteration for a fixed temperature
>
> #######
>
> dist<-function(x1,y1,x2,y2){ # computes the Euclidean distance between (x1,y1) and
(x2,y2)
+      return(sqrt(  ((x1-x2)^2) + ((y1-y2)^2)  ))
+ }
>
> #######
>
> cost<-function(ord){ #computes the total travelling distance of a order of points
+      p=length(ord)
+      d=rep(0,p)
+      for(i in 1:(p-1)){
+      d[i]=dist( xlist[ord[i]],ylist[ord[i]],xlist[ord[i+1]],ylist[ord[i+1]] )
+      }
+      d[p]<- dist( xlist[ord[1]],ylist[ord[1]],xlist[ord[p]],ylist[ord[p]] ) # the distance between
the first and last city
+      cost=sum(d)
+      return(cost)
+ }
> cost1=rep(0,M)
> numaccept=0 # to keep track of accepting
> templist=rep(0,M) # to keep track of temperature
> xlist=runif(n,0,2) # generating random points in 2 dimension as to be cities
> ylist=runif(n,0,2)
> ord=seq(1:n)  # initial order for the path
>
> #######
> for(i in 1:M){
+      cost1[i]=cost(ord)
+      for(t in 1:(fixed.itr)){
+      # propose to move
+      a=sample(1:n,2) # randomly choosing two cities
+      result=ord
+      result[a[2]]=ord[a[1]]  # swapping
+      result[a[1]]=ord[a[2]]  # swapping
+      U=runif(1)
+     if( U < exp( (cost(ord)-cost(result))/temp )){
```

```
+       ord<-result # accept/reject
+       numaccept <- numaccept + 1 # to compute the acceptance rate
+      }
+      }
+      templist[i]=temp
+      temp=temp*0.999 # update temperature
+ #    temp=tempfactor*temp
+ }
> library(calibrate)
> ord
 [1]  6  1  3  7  9  8  4 10  2  5
> cat(" iterations = ",M,"\n")
 iterations =  10000
> cat("acceptance rate = ", numaccept/(M*fixed.itr),"\n")
acceptance rate =  0.4081
> plot(xlist,ylist,xlab="X",ylab="Y",main="n=10  temp=50  M=10^4  fixed.itr=5")
> textxy(xlist,ylist,(1:n),cx=1)
> lines(xlist[ord],ylist[ord])
> lines(c(xlist[ord[1]],xlist[ord[n]]),c(ylist[ord[1]],ylist[ord[n]]),col="red")
> plot(templist,type='l',main="temperature cooling down")
> plot(cost1,type='l',main="Cost(travelling distance) for n=10")
> library(calibrate)
> ord
 [1]  2  7  9  4  5  6  3  8 10  1
> cat(" iterations = ",M,"\n")
 iterations =  10000
> cat("acceptance rate = ", numaccept/(M*fixed.itr),"\n")
acceptance rate =  0.4159
> plot(xlist,ylist,xlab="X",ylab="Y",main="n=10  temp=50  M=10^4  fixed.itr=5")
> textxy(xlist,ylist,(1:n),cx=1)
> lines(xlist[ord],ylist[ord])
> lines(c(xlist[ord[1]],xlist[ord[n]]),c(ylist[ord[1]],ylist[ord[n]]),col="red")
> plot(templist,type='l',main="temperature cooling down")
> plot(cost1,type='l',main="Cost(travelling distance) for n=10")
> library(calibrate)
> ord
 [1]  5  9  7  1  2  3  8 10  6  4
> cat(" iterations = ",M,"\n")
 iterations =  10000
> cat("acceptance rate = ", numaccept/(M*fixed.itr),"\n")
acceptance rate =  0.42914
> plot(xlist,ylist,xlab="X",ylab="Y",main="n=10  temp=50  M=10^4  fixed.itr=5")
> textxy(xlist,ylist,(1:n),cx=1)
> lines(xlist[ord],ylist[ord])
> lines(c(xlist[ord[1]],xlist[ord[n]]),c(ylist[ord[1]],ylist[ord[n]]),col="red")
> plot(templist,type='l',main="temperature cooling down")
> plot(cost1,type='l',main="Cost(travelling distance) for n=10")
```

6.3 Appendix 3

The code starts on next page

```
### Sudoku Puzzles

############### different ways of intializing with respect to the
 clue matrix

# random sample
B=clue
smpl=NULL
for(k in 1:9){
    smpl=c(smpl,rep(k,9 - sum(clue==k))) # because each digit
 should appear 9 times in the overall matrix
}
B[clue==0]=sample(smpl) # filling the matrix clue randomly with
 keeping the actual clues and having exactly 9 copies for each
 digit

# random sample, preserving the uniqueness of all digits in all
 rows
B=clue
for(i in 1:9){
    for(j in 1:9){
        a=NULL
        b=NULL
        if(clue[i,j]>0){ a=c(a,clue[i,j]) # keep tracking of the
 clues
            b=c(b,j)} # the columns of the clues
    }
    if(length(a)==0) B[i,]=sample(1:9) # if no clues, just sample
 normally
    else B[i,-b]=sample((1:9)[-a]) # if any clues, sample from the
 rest
}
B

# random sample, preserving the uniqueness of all digits in all
 blocks
B=clue
for(i in 1:9){
    r_block = seq(1,3) + (3* ( (i-1) %% 3 )) # row block index
    c_block = seq(1,3) + (3* floor( (i-1)/3 ) ) # column block
 index
    B[r_block,c_block]=sample(1:9) # normal sampling
}
B
```

```
#################

cost<- function(B){ # computing the # of repeated digits in each
 row,column,block
    count=0
    for( i in 1:9){
    r_block = seq(1,3) + (3* ( (i-1) %% 3 )) # row index for block
    c_block = seq(1,3) + (3* floor( (i-1)/3 ) ) # column index for
 block
        for(j in 1:9){
        if( sum(B[i,]==j) > 1 ) count <- count + sum(B[i,]==j) -1
 # num of repeated j in row i except itself
        if( sum(B[,i]==j) > 1 ) count <- count + sum(B[,i]==j) -1
 # num of repeated j in column i except itself
if( sum(B[r_block,c_block]==j) > 1 )count<- count +
 sum(B[r_block,c_block]==j) -1 #num of repeated j in the specified
 block
        }

    }
    return(count)

}


constr<-function(r,c,B){ # computes the # of contraints for cell
 r,c
    count=0
    if(sum(B[r,c]==B[r,])>1) count<-count + sum(B[r,c]==B[r,]) -1
    # num of cells equal to B[r,c] in its row not including itself
    if(sum(B[r,c]==B[,c])>1) count<- count + sum(B[r,c]==B[,c]) -1
    # num of cells equal to B[r,c] in its column not including
 itself
    r_block= seq(1,3) + ( 3*floor((r-1)/3) )#row index for the
 block B[r,c] is in
    c_block= seq(1,3) + ( 3*floor((c-1)/3) )#column index for the
 block B[r,c] is in
    if(sum( B[r_block,c_block]==B[r,c] ) > 1)  count<-count +
 sum( B[r_block,c_block]==B[r,c] ) -1 # num of repeated B[r,c] in
 its own block
    return(count)
}
```

```
swap<-function(B){
    probl=matrix(0,nrow=9,ncol=9)
    for(i in 1:9){
        for(j in 1:9){
            probl[i,j]=exp(constr(i,j,B)) # matrix of proportional
 probabilities for all cells
        }
    }
    swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/sum(probl))
 # choosing 2 cells at random with probability proportion to exp(i)
 with i contraints
    while(B[swap[1]]==B[swap[2]]){ # this while loop is making
 sure that the swapping is not useless
        swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/
 sum(probl))
    }
    result=B
    result[swap[1]]=B[swap[2]] #swapping
    result[swap[2]]=B[swap[1]] #swapping
    return(result)

}
cost1=rep(0,M)
numaccept=0 # keep track of acceptance
temp=70 # initial temperature
M=10^4 # num of iterations
for( i in 1:M){
    cost1[i]=cost(B)
    # updating temperatue
    temp=temp*0.999
#   for(t in 1:10){
    # proposing
    propose=swap(B)
    U=runif(1)
    if( U < exp( (cost(B)-cost(propose))/temp ) ){ # accept/reject
    B<-propose
    numaccept <- numaccept + 1 # counting the number of accepted
 moves
    }
#   }
    if(cost(B)==0) break
```

```r
}
B
cost(B)
cat("acceptance rate = ", numaccept/M,"\n")
plot(cost1,type='l',main="Cost function for Example 2")

###################### EXAMPLE 1

clue=matrix(0,ncol=9,nrow=9)
clue[1,c(6,8)]=c(6,4)
clue[2,c(1:3,8)]=c(2,7,9,5)
clue[3,c(2,4,9)]=c(5,8,2)
clue[4,3:4]=c(2,6)
clue[6,c(3,5,7:9)]=c(1,9,6,7,3)
clue[7,c(1,3:4,7)]=c(8,5,2,4)
clue[8,c(1,8:9)]=c(3,8,5)
clue[9,c(1,7,9)]=c(6,9,1)


######################  EXAMPLE 2

clue=matrix(0,ncol=9,nrow=9)
clue[1,c(1,6,7)]=c(8,1,2)
clue[2,c(2:3)]=c(7,5)
clue[3,c(5,8,9)]=c(5,6,4)
clue[4,c(3,9)]=c(7,6)
clue[5,c(1,4)]=c(9,7)
clue[6,c(1,2,6,8,9)]=c(5,2,9,4,7)
clue[7,c(1:3)]=c(2,3,1)
clue[8,c(3,5,7,9)]=c(6,2,1,9)
```

```
> ### Sudoku Puzzles
> ###################### EXAMPLE 1
>
> clue=matrix(0,ncol=9,nrow=9)
> clue[1,c(6,8)]=c(6,4)
> clue[2,c(1:3,8)]=c(2,7,9,5)
> clue[3,c(2,4,9)]=c(5,8,2)
> clue[4,3:4]=c(2,6)
> clue[6,c(3,5,7:9)]=c(1,9,6,7,3)
> clue[7,c(1,3:4,7)]=c(8,5,2,4)
> clue[8,c(1,8:9)]=c(3,8,5)
> clue[9,c(1,7,9)]=c(6,9,1)
> # random sample
> B=clue
> smpl=NULL
> for(k in 1:9){
+       smpl=c(smpl,rep(k,9 - sum(clue==k))) # because each digit should appear 9
times in the overall matrix
+ }
> B[clue==0]=sample(smpl) # filling the matrix clue randomly with keeping the actual
clues and having exactly 9 copies for each digit
> #################
>
> cost<- function(B){ # computing the # of repeated digits in each row,column,block
+       count=0
+       for( i in 1:9){
+       r_block = seq(1,3) + (3* ( (i-1) %% 3 )) # row index for block
+       c_block = seq(1,3) + (3* floor( (i-1)/3 ) ) # column index for block
+               for(j in 1:9){
+               if( sum(B[i,]==j) > 1 ) count <- count + sum(B[i,]==j) -1 # num of repeated j in
row i except itself
+               if( sum(B[,i]==j) > 1 ) count <- count + sum(B[,i]==j) -1 # num of repeated j in
column i except itself
+ if( sum(B[r_block,c_block]==j) > 1 )count<- count + sum(B[r_block,c_block]==j) -1
#num of repeated j in the specified block
+               }
+
+       }
+       return(count)
+
+ }
>
>
> constr<-function(r,c,B){ # computes the # of contraints for cell r,c
+       count=0
+       if(sum(B[r,c]==B[r,])>1) count<-count + sum(B[r,c]==B[r,]) -1
+       # num of cells equal to B[r,c] in its row not including itself
+     if(sum(B[r,c]==B[,c])>1) count<- count + sum(B[r,c]==B[,c]) -1
+     # num of cells equal to B[r,c] in its column not including itself
+     r_block= seq(1,3) + ( 3*floor((r-1)/3) )#row index for the block B[r,c] is in
+     c_block= seq(1,3) + ( 3*floor((c-1)/3) )#column index for the block B[r,c] is in
+    if(sum( B[r_block,c_block]==B[r,c] ) > 1)  count<-count +
sum( B[r_block,c_block]==B[r,c] ) -1 # num of repeated B[r,c] in its own block
+     return(count)
+ }
>
>
>
> swap<-function(B){
+     probl=matrix(0,nrow=9,ncol=9)
+     for(i in 1:9){
+           for(j in 1:9){
+                 probl[i,j]=exp(constr(i,j,B)) # matrix of proportional probabilities for all
cells
+           }
+     }
+     swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/sum(probl)) # choosing 2
cells at random with probability proportion to exp(i) with i contraints
+     while(B[swap[1]]==B[swap[2]]){ # this while loop is making sure that the swapping
is not useless
+           swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/sum(probl))
+     }
+     result=B
+     result[swap[1]]=B[swap[2]] #swapping
+     result[swap[2]]=B[swap[1]] #swapping
+     return(result)
+
+ }
>
> temp=70 # initial temperature
> M=10^4 # num of iterations
> for( i in 1:M){
+     # updating temperature
+     temp=temp*0.999
+ #   for(t in 1:10){
+     # proposing
+     propose=swap(B)
+     U=runif(1)
+     if( U < exp( (cost(B)-cost(propose))/temp ) ) # accept/reject
+     B<-propose
+ #   }
+     if(cost(B)==0) break
+
+ }
```

```
> B
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    8    3    5    2    6    7    4    9
 [2,]    2    7    9    4    1    3    8    5    6
 [3,]    4    5    6    8    7    9    3    1    2
 [4,]    7    3    2    6    4    1    5    9    8
 [5,]    9    6    8    3    5    7    1    2    4
 [6,]    5    4    1    8    9    2    6    7    3
 [7,]    8    1    5    2    3    9    4    6    7
 [8,]    3    9    7    1    6    4    2    8    5
 [9,]    6    2    4    7    8    5    9    3    1
> cost(B)
[1] 2
>
> B=clue
> smpl=NULL
> for(k in 1:9){
+       smpl=c(smpl,rep(k,9 - sum(clue==k))) # because each digit should appear 9
times in the overall matrix
+ }
> B[clue==0]=sample(smpl) # filling the matrix clue randomly with keeping the actual
clues and having exactly 9 copies for each digit
> temp=70 # initial temperature
> M=10^4 # num of iterations
> for( i in 1:M){
+       # updating temperature
+       temp=temp*0.999
+ #     for(t in 1:10){
+       # proposing
+       propose=swap(B)
+       U=runif(1)
+       if( U < exp( (cost(B)-cost(propose))/temp ) ) # accept/reject
+       B<-propose
+ #     }
+       if(cost(B)==0) break
+
+ }
> B
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    3    8    5    2    6    7    4    9
 [2,]    2    7    9    3    4    1    8    5    6
 [3,]    4    5    6    8    7    9    3    1    2
 [4,]    7    4    2    6    3    5    1    9    8
 [5,]    9    6    3    1    8    7    5    2    4
 [6,]    5    8    1    4    9    2    6    7    3
 [7,]    8    9    5    2    1    3    4    6    7
 [8,]    3    1    7    9    6    4    2    8    5
 [9,]    6    2    4    7    5    8    9    3    1
> cost(B)
[1] 0
>
```

```
> ### Sudoku Puzzles
>
> #####################  EXAMPLE 2
>
> clue=matrix(0,ncol=9,nrow=9)
> clue[1,c(1,6,7)]=c(8,1,2)
> clue[2,c(2:3)]=c(7,5)
> clue[3,c(5,8,9)]=c(5,6,4)
> clue[4,c(3,9)]=c(7,6)
> clue[5,c(1,4)]=c(9,7)
> clue[6,c(1,2,6,8,9)]=c(5,2,9,4,7)
> clue[7,c(1:3)]=c(2,3,1)
> clue[8,c(3,5,7,9)]=c(6,2,1,9)
>
> # random sample
> B=clue
> smpl=NULL
> for(k in 1:9){
+     smpl=c(smpl,rep(k,9 - sum(clue==k))) # because each digit should appear 9
times in the overall matrix
+ }
> B[clue==0]=sample(smpl) # filling the matrix clue randomly with keeping the actual
clues and having exactly 9 copies for each digit
>
> ################
>
> cost<- function(B){ # computing the # of repeated digits in each row,column,block
+     count=0
+     for( i in 1:9){
+     r_block = seq(1,3) + (3* ( (i-1) %% 3 )) # row index for block
+     c_block = seq(1,3) + (3* floor( (i-1)/3 ) ) # column index for block
+         for(j in 1:9){
+         if( sum(B[i,]==j) > 1 ) count <- count + sum(B[i,]==j) -1 # num of repeated j in
row i except itself
+         if( sum(B[,i]==j) > 1 ) count <- count + sum(B[,i]==j) -1 # num of repeated j in
column i except itself
+ if( sum(B[r_block,c_block]==j) > 1 )count<- count + sum(B[r_block,c_block]==j) -1
#num of repeated j in the specified block
+         }
+
+     }
+     return(count)
+
+ }
>
>
> constr<-function(r,c,B){ # computes the # of contraints for cell r,c
```

```
+      count=0
+      if(sum(B[r,c]==B[r,])>1) count<-count + sum(B[r,c]==B[r,]) -1
+      # num of cells equal to B[r,c] in its row not including itself
+      if(sum(B[r,c]==B[,c])>1) count<- count + sum(B[r,c]==B[,c]) -1
+      # num of cells equal to B[r,c] in its column not including itself
+      r_block= seq(1,3) + ( 3*floor((r-1)/3) )#row index for the block B[r,c] is in
+      c_block= seq(1,3) + ( 3*floor((c-1)/3) )#column index for the block B[r,c] is in
+     if(sum( B[r_block,c_block]==B[r,c] ) > 1)  count<-count +
sum( B[r_block,c_block]==B[r,c] ) -1 # num of repeated B[r,c] in its own block
+      return(count)
+ }
>
>
>
> swap<-function(B){
+      probl=matrix(0,nrow=9,ncol=9)
+      for(i in 1:9){
+             for(j in 1:9){
+                    probl[i,j]=exp(constr(i,j,B)) # matrix of proportional probabilities for all
cells
+             }
+      }
+      swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/sum(probl)) # choosing 2
cells at random with probability proportion to exp(i) with i contraints
+      while(B[swap[1]]==B[swap[2]]){ # this while loop is making sure that the swapping
is not useless
+             swap=sample((1:81)[clue==0],2,prob=probl[clue==0]/sum(probl))
+      }
+      result=B
+      result[swap[1]]=B[swap[2]] #swapping
+      result[swap[2]]=B[swap[1]] #swapping
+      return(result)
+
+ }
> cost1=rep(0,M)
> numaccept=0 # keep track of acceptance
> temp=70 # initial temperature
> M=10^4 # num of iterations
> for( i in 1:M){
+      cost1[i]=cost(B)
+      # updating temperatue
+      temp=temp*0.999
+ #   for(t in 1:10){
+      # proposing
+      propose=swap(B)
+      U=runif(1)
+      if( U < exp( (cost(B)-cost(propose))/temp ) ){ # accept/reject
+      B<-propose
+      numaccept <- numaccept + 1 # counting the number of accepted moves
+      }
+ #    }
+      if(cost(B)==0) break
+
+ }
> B
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    8    9    4    6    7    1    2    3    5
 [2,]    6    7    5    3    4    2    8    9    1
 [3,]    3    1    2    9    5    8    7    6    4
 [4,]    1    4    7    2    3    4    9    8    6
 [5,]    9    6    3    7    8    5    5    1    2
 [6,]    5    2    8    1    6    9    3    4    7
 [7,]    2    3    1    5    9    6    4    7    8
 [8,]    4    5    6    8    2    3    1    5    9
 [9,]    7    8    9    4    1    7    6    2    3
> cost(B)
[1] 4
> cat("acceptance rate = ", numaccept/M,"\n")
acceptance rate =  0.3745
> plot(cost1,type='l',main="Cost function for Example 2")
>
> # random sample
> B=clue
> smpl=NULL
> for(k in 1:9){
+      smpl=c(smpl,rep(k,9 - sum(clue==k))) # because each digit should appear 9
times in the overall matrix
+ }
> B[clue==0]=sample(smpl) # filling the matrix clue randomly with keeping the actual
clues and having exactly 9 copies for each digit
> cost1=rep(0,M)
> numaccept=0 # keep track of acceptance
> temp=70 # initial temperature
> M=10^4 # num of iterations
> for( i in 1:M){
+      cost1[i]=cost(B)
+      # updating temperatue
+      temp=temp*0.999
+ #    for(t in 1:10){
+      # proposing
+      propose=swap(B)
+      U=runif(1)
+      if( U < exp( (cost(B)-cost(propose))/temp ) ){ # accept/reject
+      B<-propose
+      numaccept <- numaccept + 1 # counting the number of accepted moves
+      }
+ #   }
+      if(cost(B)==0) break
+
+ }
> B
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    8    4    9    6    7    1    2    5    3
 [2,]    6    7    5    4    2    3    9    1    8
 [3,]    3    1    2    8    5    9    7    6    4
 [4,]    1    8    7    2    3    4    5    9    6
 [5,]    9    6    4    7    8    5    3    2    1
 [6,]    5    2    3    1    6    9    8    4    7
 [7,]    2    3    1    9    4    8    6    7    5
 [8,]    4    5    6    3    2    7    1    8    9
 [9,]    7    9    8    5    1    6    4    3    2
> cost(B)
[1] 2
> cat("acceptance rate = ", numaccept/M,"\n")
acceptance rate =  0.3787
> plot(cost1,type='l',main="Cost function for Example 2")
>
>
```