Simulated Annealing

Jeremy Morris

December 2, 2005

In order to demonstrate that the code is working, I have created the plots in Figure 1. The plots show what happens for different choices of β_n for the Traveling Salesman problem with 4 cities. The code is written to handle the general case and even allows for non-symmetric distance matrices.

The distance matrix is defined using exponential random numbers with mean 700, that are then converted to integers. This turns out to make it easier to determine if the true minimum has been reached.

The graphs show how many times a certain path was chosen. We are supposed to see the uniform distribution on the set of minimizers, in this case the paths of minimum distance.

There are two main difficulties. The first is how to choose ε and n. The second is the time it takes to generate the graphs in order to tell if the method is working or not. In the graphs that have been displayed, the values are chosen as $\varepsilon = 0.01, n = 10,000$. This could take up to one hour, just for the 4 city case. The values for ε and n have been chosen using trial and error. If we were to increase the number of cities to 10, for example, the time it would take to see if it worked would be very high. This appears to be the main downfall of solving the Traveling Salesman problem this way. There is no theoretical way of knowing how to chose the values of ε and n to make it practical for a large number of cities. It is actually quicker and more accurate to use the computer to compute the answer explicitly.

Another difficulty here is that with only 4 cities, there are really no local minima like there would be with a larger number, say 10 or more, so we cannot really check if in the case where $\beta_n = \varepsilon n$, the method gets stuck in those local minima.

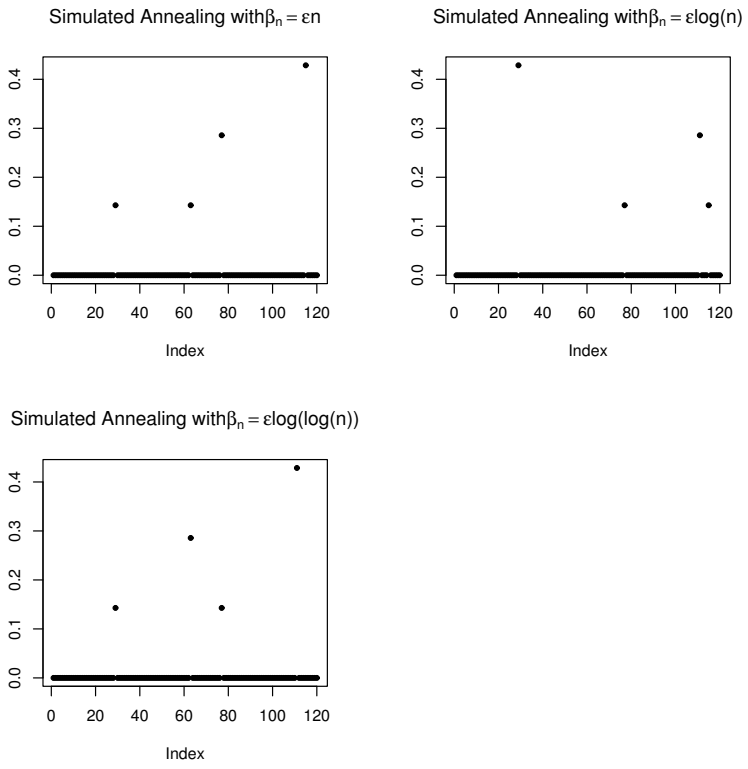

Figure 1: Graphs for 4 city Traveling Salesman

```
# Simulated Annealing for Traveling Salesman

pathPermutations <- function(sigma){
    l <- length(sigma)
    num <- choose(l,2) + 1

    r <- matrix(0,nrow=num,ncol=l)

    c <- 1
    for( i in 1:(l-1) ){
       for( j in (i+1):l ){
          r[c,] <- sigma
          r[c,i] <- sigma[j]
          r[c,j] <- sigma[i]
          c <- c + 1
       }
    }
    r[c,] <- sigma

    return(r)
}
```

```
H <- function(path,D){
    l <- length(path)
    d <- D[path[l],path[1]]
    for( i in 2:(l) ){
       d <- d + D[path[i-1],path[i]]
    }
    return(d)
}

allPn <- function(s){
    library(combinat)
    l <- length(s)
    n <- factorial(l)
    t(matrix(unlist(permn(sort(s))),ncol=n,nrow=l))
}

pi.term <- function(h.i,h.j,beta){
    r <- exp( -beta * (h.j - h.i) )
    if( h.j - h.i == 0 ){
       r <- 1
    }
    return(r)
}

nextPath <- function(original,beta,D){
    l <- length(original)
    n <- factorial(l)
    paths <- pathPermutations(original)


    lpt <- choose(l,2) + 1
    p <- numeric(lpt)

    this <- lpt
    c <- 1

    for( i in 1:(lpt-1) ){
       pi_i <- sDist(beta,paths[this,],D)
       pi_j <- sDist(beta,paths[i,],D)

       ratio <- pi.term(H(paths[this,],D),H(paths[i,],D),beta)

       p[i] <- 1 / lpt

       if( ratio < 1 ){
          p[i] <- (1/lpt) * ratio
       }

    }
```

```
    p[this] <- 1 - sum(p)

    ranU <- runif(1)
    c <- 1
    rtotal <- p[1]

    while( rtotal < ranU ){
       c <- c + 1
       rtotal <- rtotal + p[c]
    }

    return(paths[c,])
}
```