Analysis

Contents

Objectives

A short description of the projects main problem is available on our home page. We are tasked with writing a simulation that choose large n and m that we can sample the expected travel time of m-n+1 circuits which we are asked to show is approximately the true expected travel time of a circuit chosen uniformly

$$\sum_i f(i)\mu(i) \approx \frac{1}{m+1}\sum_{k=0}^{m} f(X_k)$$

. Towards this goal we first labeled 20 cities numerically from 1:20

```
U <-c(1:20) #vertices
```

In graph terminology these are our vertices and will make up the elements within our state space. Originally we had proceeded with the idea that we can sample from a generator matrix that would simulate all the Hamiltonian circuits but this would require non-existent computing power to generate a 20! by 20! matrix.

Thus its best to consider the state space as a collection of all the valid circuits of which there are

```
## [1] 2.432902e+18
```

Thus the generator isn't a matrix but instead a method which will choose two cities uniformly at random. Before writing the method, we first construct the edges as per the requirement. To this end we first constructed a 20 by 20 distance matrix D and populated it using a uniform(0,1) density.

Distance Matrix

```
D <- matrix(runif(n=20*20, min=0, max =1), ncol = 20) #uniformly distributed distance matrix
```

We then manually specified 5 pairs (i,j) for which the entry in the distance matrix D would be zero implying travel restriction.We then transform D into a symmetric matrix by assigning the lower triangular of the matrix the entries of the upper triangular of the transpose of D.Finally we set the diagonal entries to zero as its assumed the distance from city i to i is 0. Thus as you can see there are 185 pairs with positive densities (19*20)/2 - 5 from the matrix or $\binom{20}{2} - 5$.

```
D[4,6] = 0
D[9,13]= 0
D[1,19]= 0
D[10,17]= 0
D[3,8] = 0 #NO TRAVEL NO DISTANCE
D[lower.tri(D)] = t(D)[lower.tri(D)] #symmetric matrix
diag(D) <- rep(0,20) #diagonals are 0
```

We note that the forbidden pairs must be entered in the form of D[i,j] where i<j or else when we go to make the matrix symmetric the upper triangular entry is still most likely non-zero. This can be remedied by manually make all 10 entries 0, or letting j<i and assigning the upper-triangular of D with the lower triangular of the transpose of D or in a proper application handling user input.

Algorithm

Thus with the vertices set up and the weighted-edges specified, we have initialized our graph. We can now consider our Q. As mentioned above, the most prudent method as suggested was a method which picks two points uniformly at random and intechanges them. The code chunck below is the function that we desire.

```
tspMCMC <- function(n,m){
S <- list(c(1:20)) #preferably this sequence will be randomly generated and a proper circuit.
rejection_counter <- 0
i <- 1
j <- 1
travel_time <- rep(0, m-n+1)
while(i <= m){
  P <- sample(1:20,size=1) #basic sampling
  Q <- sample(1:20,size=1) #likewise for proposed state, additionally Q <- sample(1:20,size=1, prob =D[
  index_p = match(P, unlist(S[i])) #the position of P in the sequence
  index_q = match(Q, unlist(S[i])) #likewise for the second element in the pair
  k <- unlist(S[i])
  if(indicatorPermissible(k,index_p,index_q) == 1){
    if(i >= n){
      travel_time[j] = distance_travelled(S[i])
      j <- j + 1
    }
    S[i+1] <- list(c(replace(k, c(index_p,index_q), c(Q,P)))) #adds a new sequence to S interchanging P
    i <- i + 1
    #return(S[i+1])
  }
  else{
    rejection_counter = rejection_counter+1 #see how many time our proposed transition introduced forbi
  }
}
   answer = mean(travel_time)*(m-n+1)/(m+1)
   print(length(travel_time))
   print(rejection_counter)
   return(answer)
}
```

There is alot to unpack but very concisely it takes two arguments n(integer) and m(integer). It then samples two points, extracts their place in the sequence and pass it to along with the sequence to another function

called *indicatorPermissable*. It's the responsibility of the *indicatorPermissable* to return 0 if the proposed interchange creates a sequence where travel between two forbidden cities is introduced. In the case where the indicator returns 1 we accept the new sequence and add it to the list of valid circuits that we have sampled.Furthermore if we have already sampled n **valid** circuits we also start computing the travel times by passing the sequence to a function called *distance_travelled*.

One decision of importance that was taken was the use of a while loop instead of a for loop. This is done for multiple reasons one of which is that it gives up greater control of our simulation. It allows us to sample m unique circuits without repetitions. For example, if there is a proposed interchange that is rejected we don't add it to the list of valid sequences we have sampled. This is done so that we don't have to compute the travel time for a sequence multiple times in the case where we have rejected an interchange.

Observations

Now let us test the algorithm using the same forbidden cities from above and a Uni(0,1) distributed distance matrix, additionally choose $n = 20,000$ and $m = 40,000$

```
tspMCMC(20000,40000)
```

```
## Rejection Counter:  12006
## Average Return Time: 5.028794 seconds
## Computational Length: 1.840192 seconds
```

Furthermore keeping the distance matrix fixed we can observe the results for multiple choices of n and m.

```
tspMCMC(20000,40000)
tspMCMC(40000,80000)
tspMCMC(80000,160000)
tspMCMC(160000,320000)
```

```
## tspMCM( 20000 , 40000 )
## Rejection Counter:  12300
## Average Return Time: 4.988946 seconds

## tspMCM( 40000 , 80000 )
## Rejection Counter:  24160
## Average Return Time: 4.994616 seconds

## tspMCM( 80000 , 160000 )
## Rejection Counter:  48507
## Average Return Time: 4.983314 seconds

## Computational Length: 10.50478 minutes
```

Appendix

The following R.file contains our code