MATH 4330/6602

Stochastic Processes

Metropolis Algorithm for expectations

December 2, 2020

- Quynh Vu 216714271
- Raghad Ibrahim 215376395
- Maninder Dhaliwal 214998991
- Mehrdad Kazemi 216057754

Written in LaTeX

Department of Mathematics and Statistics
York University
Canada

Contents

Acknowledgments

Our team would like to take the time to thank Professor Salisbury on providing this topic for productive academic discussions and research for a final project.

1. Instructions:

1.1 Introduction

The Metropolis algorithm gives a way of producing samples from a target probability distribution μ on a discrete state space U with all $\mu_i > 0$. It is used in situations where it is difficult or impossible to directly sample from μ (the way you would if you were generating random numbers from a normal distribution). It uses an irreducible transition matrix Q on U that is symmetric (i.e., $Q_{ij} = Q_{ji}$) for each i, j. Any such Q will work, but the art of using Metropolis lies in choosing a Q that is likely to make the algorithm work quickly. We define new transition probabilities as follows:

$$p_{ij} = \begin{cases} q_{ij} min(1, \frac{\mu_j}{\mu_i}) \text{ if j} \neq \text{i} \\ 1 - \sum_{k \neq i} q_{ik} min(1, \frac{\mu_k}{\mu_i}) \text{ if j = i} \end{cases}$$

This looks complicated, so let me describe it a dynamic way:

- Starting from i, we use Q to propose a state j to move to. If $\mu_j \geq \mu_i$ then accept that move. If $\mu_j < \mu_i$, then we flip a biased coin so that we accept the move with probability $\frac{\mu_j}{\mu_i} < 1$ and reject the move otherwise (and just stay at state i). Or, what amounts to the same thing, generate a uniform random number U on [0, 1] and accept the move if U $\leq \frac{\mu_j}{\mu_i}$.
- For example, if all the μ_i's are equal, then the target distribution is uniform on U. In that case, P = Q and there is no rejection. Because Q is symmetric, it is reversible with respect to the uniform distribution, so μ is an invariant distribution. You may find it strange to think that sampling from a uniform distribution could be hard, but if U is a large complicated set (so large that we cannot count its elements easily), this may in fact be the case.
- Note that we do not actually need to know the values of μ_i, but just the ratios $\frac{\mu_j}{\mu_i}$. So in the uniform case, we know these ratios is equal to 1, even if we cannot count the size of U precisely in order to find $\mu_i = \frac{1}{\#U}$

1.2 Travelling Salesman Problem

Suppose we have N cities located at the vertices of a graph. The graph has an edge between cities if it is possible to travel between them, and we label the edge with the travel time. A (Hamiltonian) circuit is a sequence of cities that starts and ends at some city, visits that city only at the beginning and end, and visits every other city exactly once. The classic Travelling Salesman Problem is to find the shortest such circuit. When N is large, this is hard. Our problem will be a bit different: I want you to find the average travel time of circuits. In other words, the expected travel time for a circuit chosen uniformly from all circuits. Use Metropolis with $\mu = 1$ (so no rejection)

2. Report:

In many applications, the full acknowledgement of complexity and structure are difficult to obtain and requires specific methodologies. We hence seek an alternative to coerce the problem into a simpler framework of an available methods. For that reason, the Markov chain Monte Carlo (MCMC) methods are popolar in real-world problems as they provide enormous scope for realistic statistical modelling as well as a unifying framework within which many complex problems can be analyzed using generic software.

MCMC methodologies are employed when sampling from possibly high-dimensional probability distributions is needed to make inferences about model parameters or to make predictions. It essentially draws samples from a target distribution by running a well-constructed Markov chain for a long time and then forms sample averages to approximate expectations. Such chains are constructed using the general framework of Metropolis Hastings algorithm. The focus of the project is on the Metropolis method rather than its generalization Metropolis-Hastings method where the proposal matrix Q is not symmetric.

In particular, if we start with a state-space S, and an invariant probability distribution $\{\mu_i\}$ on S which we want to sample from, the Metropolis algorithm designs a Markov chain that proceeds in two stages. Initially, a new state is proposed from a proposal transition matrix $Q = \{q_{ij} : i, j \in S\}$. In the successive stage, the proposed state is either accepted or rejected. If it is accepted, then the Markov chain moves there, but if it is rejected, the chain stays where it is.

2.1 P is reversible with respect to an invariant probability distribution μ

By definition of reversibility, matrix P is reversible with respect to μ if and only if $\mu_i p_{ij} = \mu_j p_{ji}$ for every i and j.

Assume that the symmetric and irreducible proposal matrix Q is at hand.

- If i $\neq$ j, then $\mu_i p_{ij} = \mu_i \Big[q_{ij} min(1; \frac{\mu_j}{\mu_i})\Big] = q_{ij} \Big[\mu_i min(1; \frac{\mu_j}{\mu_i})\Big] = q_{ij} min(\mu_i, \mu_j)$.

  Likewise, $\mu_j p_{ji} = \mu_j \Big[q_{ji} min(1; \frac{\mu_i}{\mu_j})\Big] = q_{ji} \Big[\mu_j min(1; \frac{\mu_i}{\mu_j})\Big] = q_{ji} min(\mu_j, \mu_i) = q_{ji} min(\mu_i, \mu_j)$.

  As matrix Q is symmetric, it follows that $q_{ij} = q_{ji}$. That is, $q_{ij} min(\mu_i, \mu_j) = q_{ji} min(\mu_i, \mu_j)$ or $\mu_i p_{ij} = \mu_j p_{ji}$

- If j = i, then $p_{ij} = 1 - \sum_{k \neq i} q_{ik} min(1, \frac{\mu_k}{\mu_i})$. Thus, $\mu_i p_{ij} = \mu_i \Big[1 - \sum_{k \neq i} q_{ik} min(1, \frac{\mu_k}{\mu_i})\Big]$.

  Since i = j, it follows that $\mu_i = \mu_j$ and q_{ik} for some k $\neq$ i is equivalent to q_{jk} for some k $\neq$ j. It is trivial that $\mu_i p_{ij} = \mu_j \Big[1 - \sum_{k \neq j} q_{jk} min(1, \frac{\mu_k}{\mu_j})\Big] = \mu_j p_{ji}$

We conclude that $\mu_i p_{ij} = \mu_j p_{ji}$ for every i and j, and P is reversible with respect to μ. Thus, μ is the invariant probability distribution.

2.2 P is irreducible, and that if μ is not perfectly uniform then P is aperiodic.

Assume that the symmetric and irreducible proposal matrix Q is at hand.

Since Q is irreducible, each rate q_{ij} can propose the transition from i to j for every i and j. The transition probability matrix P is defined based on the proposal matrix Q by $p_{ij} = \begin{cases} q_{ij} min(1, \frac{\mu_j}{\mu_i}) \text{ if j} \neq \text{i} \\ 1 - \sum_{k \neq i} q_{ik} min(1, \frac{\mu_k}{\mu_i}) \text{ if j = i} \end{cases}$

Hence, the chain can move from state i to state j with probability p_{ij} for every i and j. Thus, the P matrix is irreducible. The irreducibility of P follows from the irreducibility of Q.

The period of state i is defined by period(i) = $gcd\{n \geq 1 : p_{ii}^{(n)}>0\}$. The chain is aperiodic if period(i) = 1 for every i. When μ is not perfectly uniform, there is rejection since $min(1, \frac{\mu_k}{\mu_i})$ is no longer 1 as in the uniform case where $\mu_i = \mu_j$. If $\mu_i > \mu_j$, the transition is rejected, and the chain stays put. Thus, $p_{ii}^{(1)}>0$. Since the greatest common divisor of 1 and any number n $\in \mathbb{N}$ is always 1, we do not need to consider the cases of $\mu_i < \mu_j$.

By definition, period(i) = $gcd\{1, n\}$ = 1 for every state i and n $\in \mathbb{N}$. Hence, if μ is not perfectly uniform then P is aperiodic.

2.3 Travelling Salesman Problem

For the convenience of readers, we number different states from 1 to 20!.

Let $\vec{X}_k = (a_1, a_2, ..., a_{20})$ denote the k^{th} state and S denote the state-space of $\vec{X}_k$. It is worth noting that $\mid S \mid = 20!$ circuits.

Let T_k denote the travel time to finish circuit $\vec{X}_k$ and $\vec{T} = (T_1, T_2, ..., T_{20!})$

If $\vec{T} \sim \mu$, then the average travel time is $I = \sum_{k \in S} f(T_k)\mu_k = E\big[f(T_k)\big]$ where f is a real-valued function on S. This is not feasible to compute as $\mid S \mid = 20!$. We hence opted to use the MCMC method due to the following fact:

- The simple or crude Monte Carlo estimator is $\hat{I} = \frac{\sum_{k=0}^{m} f(T_k)}{m+1}$
- $E(\hat{I}) = E\Big[\frac{\sum_{k=0}^{m} f(T_k)}{m+1}\Big] = \frac{1}{m+1}\sum_{k=0}^{m} E[f(T_k)] = \frac{1}{m+1}(m+1)E\big[f(T_k)\big] = E\big[f(T_k)\big] = I$

Thus, $\hat{I}$ is an unbiased estimator of I, and we can employ the MCMC method by running the chain through different states sufficiently long and dividing the sum of time spent in the entire process by the total of valid chains (repetitions will be removed). When m is also large, $\sum_{k \in S} f(T_k)\mu_k \approx \frac{\sum_{k=0}^{m} f(T_k)}{m+1}$.

a Coding algorithm and R output

As the length of the period an individual spends in a circuit is proportional to the sum of distances between cities in that circuit, the average time in each circuit is also the average travel distance.

(1) Label 20 cities as 1, 2, ..., 20.

```
> # Variable declaration
> N = 20      # Number of cities
> p = 5       # Number of disconnected pairs
> m = 10000 # Number of circuits needed to break in equilibrium
> n = 10000 # Number of circuits taken into the calculation of average time spent.
> M = n + m # Total number of runs
> lamda = 1 # Exponential parameter
> x = 0; y = 0; z = 0;
> # (1) Label 20 cities as 1, 2, . . . , 20
> cities <- c(1:N)
> cities
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

(2) Pick 5 pairs of cities {i, j} for which direct travel between i and j will be forbidden (e.g., because of border restrictions or because there are no flights between those cities).

The five randomly uniformly chosen pairs of cities whose direct travel route is forbidden are (18, 11), (3, 13), (12, 15), (6, 7), and (4, 13).

```
> # (2) Pick 5 pairs of cities {i, j} for which direct travel between i and j is forbidden
> # Number of pairs of cities
> pairs <- dim(combn(N, 2))[2]
> pairs
[1] 190
> for(a in 1:p){
+   x[a] = rdunif(1, 1, N);
+   y[a] = rdunif(1, 1, N)
+   for(b in 1:(a-1)){
+     if(isTRUE(x[a] != y[a] || y[a] != y[a - b] && x[a] != x[a - b])){a = a + 1}
+     else{a = a + 0}
+   }
+ }
> disconnected.pairs <- data.frame(x, y)
> disconnected.pairs #Random disconnected pairs
   x  y
1 18 11
2  3 13
3 12 15
4  6  7
5  4 13
```

(3) Generate random positive distances between the other 185 pairs of cities {i, j} with i $\neq$ j (and explain why is it 185). You should generate these using random sampling from some distribution on the positive reals, with a density.

There are $\binom{20}{2}$ = 190 pairs of cities in total. As the direct routes of five pairs of cities are forbidden, there are 190 - 5 = 185 distances to generate.

Sampling from distribution on the positive reals with a density was carried out by the Inverse cumulative distribution function method. In particular, distances between cities follow Exponential(λ) distribution ($\lambda = 1$ was used in the output).

- Matrix of U(0, 1) entries is initially generated. As every pair of cities without prohibited is connected by an edge, it implies distance from city a_i to city a_j is similar to that from city a_j to city a_i. Thus, the entry distance(a_i, a_j) = distance(a_j, a_i)
- Suppose distance D $\sim$ Exp(λ) $\Rightarrow f_D(d) = \begin{cases} \lambda e^{-\lambda d} (d \geq 0) \\ 0 (d < 0) \end{cases}$ and $F_D(d) = \begin{cases} 0 (d < 0) \\ 1 - e^{-\lambda d} (d \geq 0) \end{cases}$
- Let $U = F_D(d) = 1 - e^{-\lambda d}$

  $\Rightarrow D = F_X^{-1}(u) = \frac{ln(1-U)}{\lambda}$ or $\frac{-ln(U)}{\lambda}$ since U $\sim$ U(0,1) is equivalent to 1 - U $\sim$ U(0,1)

  $\Rightarrow D = \frac{-ln(u)}{\lambda} \sim$ Exp(λ) (0 < u $\leq$ 1).

  This proved the formula used in the code to transform U(0, 1) random variable to Exponential(λ) random variable.

The entries on the diagonal of the distance matrix are 0 as the distance(a_i, a_i) = 0 and 0 where the positioning of five disconnected pairs of cities are.

```
> # (3) Generate random positive distances between the other 185 pairs of cities
> unif.matrix <- matrix(runif(N*N), N)          # Initiate U(0,1) random variables
> distance = - (1/lamda)*log(unif.matrix)       # Inverse method - Sampling from Exp(1)
> ind <- lower.tri(distance)
> distance[ind] <- t(distance)[ind]             # d(i,j) = d(j,i)
```

```
> hist(distance,freq=F,main="Exponential(1) from Uniform by Inverse method")
> for(i in 1:N){
+   for(j in 1:N){
+     for(k in 1:p){
+       if(isTRUE(i == j)){distance[i, j] = 0} # d(i,i) = 0
+       else if(isTRUE(distance[i, j] == distance[disconnected.pairs$x[k], disconnected.pairs$y[k]])){
    distance[i, j] = 0} #distance of disconnected pairs is 0
+       else if(isTRUE(distance[i, j] == distance[disconnected.pairs$y[k], disconnected.pairs$x[k]])){
    distance[i, j] = 0}
+       else{distance[i, j] = distance[i, j]}
+     }
+   }
+ }
> distance
           [,1]      [,2]       [,3]       [,4]       [,5]       [,6]      [,7]
 [1,] 0.0000000 0.9320832 3.40274550 3.46224612 1.19654502 4.33492769 0.7942448
 [2,] 0.9320832 0.0000000 2.95075458 1.44443886 1.09131071 0.73079749 1.0592283
 [3,] 3.4027455 2.9507546 0.00000000 0.30930354 0.22677580 0.28426110 0.9469675
 [4,] 3.4622461 1.4444389 0.30930354 0.00000000 1.33021511 0.51727744 0.5014502
 [5,] 1.1965450 1.0913107 0.22677580 1.33021511 0.00000000 1.99341734 0.1447603
 [6,] 4.3349277 0.7307975 0.28426110 0.51727744 1.99341734 0.00000000 0.0000000
 [7,] 0.7942448 1.0592283 0.94696753 0.50145022 0.14476031 0.00000000 0.0000000
 [8,] 1.1244644 0.2281960 0.10145908 0.26292282 1.74954203 0.09861964 0.1207718
 [9,] 0.8629289 2.3091413 0.58792900 1.26063553 1.20324362 1.62484656 1.8825765
[10,] 1.2216259 0.1256919 0.62192313 0.05817477 0.52160036 0.96048255 0.2067408
[11,] 4.8816918 1.1170396 0.09190522 1.72337172 0.14317172 0.82204179 0.8874243
[12,] 0.9206982 2.6393456 0.70928082 0.35307245 0.08274546 1.43934647 0.8618456
[13,] 0.4997643 1.2920600 0.00000000 0.00000000 1.01664590 0.47498643 0.2491414
[14,] 0.6893786 0.2111088 2.17733081 0.12095076 0.06188887 0.83746805 0.7790525
[15,] 1.2368151 0.4316111 0.79492006 0.82426309 0.61900410 3.06729131 2.8378375
[16,] 0.3563493 2.1745272 1.86694350 2.20856169 1.10242631 0.30689120 0.9261596
[17,] 0.4986400 0.4457501 0.38334731 3.62104504 0.12261876 1.42360996 0.8669298
[18,] 0.3775854 0.3068815 1.84463904 1.07203339 0.62638842 0.39928895 0.6919842
[19,] 0.1945492 1.6278675 1.29918420 0.73060283 1.45640074 0.19685498 0.1121259
[20,] 1.8902556 0.9882417 0.91098648 2.26291030 0.84980238 2.89112853 0.5954744
             [,8]        [,9]       [,10]      [,11]      [,12]      [,13]
 [1,] 1.124464387 0.862928937 1.221625865 4.88169180 0.92069818 0.49976433
 [2,] 0.228195999 2.309141264 0.125691902 1.11703963 2.63934560 1.29206002
 [3,] 0.101459078 0.587929002 0.621923125 0.09190522 0.70928082 0.00000000
 [4,] 0.262922822 1.260635529 0.058174769 1.72337172 0.35307245 0.00000000
 [5,] 1.749542030 1.203243616 0.521600365 0.14317172 0.08274546 1.01664590
 [6,] 0.098619638 1.624846557 0.960482553 0.82204179 1.43934647 0.47498643
 [7,] 0.120771773 1.882576495 0.206740835 0.88742434 0.86184556 0.24914142
 [8,] 0.000000000 0.008268468 1.112449497 0.20920833 0.35983032 0.14252293
 [9,] 0.008268468 0.000000000 0.241027255 1.10982971 4.39937878 0.09680876
[10,] 1.112449497 0.241027255 0.000000000 2.05058038 0.99975724 0.04263172
[11,] 0.209208328 1.109829711 2.050580379 0.00000000 0.28100945 0.20604623
[12,] 0.359830323 4.399378779 0.999757235 0.28100945 0.00000000 2.51210366
[13,] 0.142522934 0.096808762 0.042631720 0.20604623 2.51210366 0.00000000
[14,] 0.509996908 2.401307280 1.304184752 1.65884084 1.05155039 0.60596072
[15,] 1.300525556 0.362436572 0.843407893 0.20105844 0.00000000 1.42252961
[16,] 0.490108875 0.553206512 0.003032895 0.80726358 0.24728912 3.44359653
[17,] 1.721092549 0.941690643 0.212812506 0.28124754 0.39311407 0.54847576
[18,] 0.532050442 1.034818940 0.771799376 0.00000000 1.13021575 0.83395115
[19,] 0.586840540 0.566059633 0.214771800 0.46358745 0.16270384 0.34840708
[20,] 1.298853877 2.573312048 0.489230036 1.37969003 2.45447300 0.74702202
           [,14]     [,15]       [,16]      [,17]      [,18]      [,19]      [,20]
 [1,] 0.68937859 1.2368151 0.356349343 0.49863997 0.37758541 0.19454925 1.89025555
 [2,] 0.21110877 0.4316111 2.174527160 0.44575015 0.30688152 1.62786750 0.98824171
 [3,] 2.17733081 0.7949201 1.866943499 0.38334731 1.84463904 1.29918420 0.91098648
 [4,] 0.12095076 0.8242631 2.208561691 3.62104504 1.07203339 0.73060283 2.26291030
 [5,] 0.06188887 0.6190041 1.102426311 0.12261876 0.62638842 1.45640074 0.84980238
 [6,] 0.83746805 3.0672913 0.306891199 1.42360996 0.39928895 0.19685498 2.89112853
 [7,] 0.77905250 2.8378375 0.926159552 0.86692981 0.69198420 0.11212590 0.59547445
 [8,] 0.50999691 1.3005256 0.490108875 1.72109255 0.53205044 0.58684054 1.29885388
 [9,] 2.40130728 0.3624366 0.553206512 0.94169064 1.03481894 0.56605963 2.57331205
[10,] 1.30418475 0.8434079 0.003032895 0.21281251 0.77179938 0.21477180 0.48923004
```

```
[11,] 1.65884084 0.2010584 0.807263581 0.28124754 0.00000000 0.46358745 1.37969003
[12,] 1.05155039 0.0000000 0.247289119 0.39311407 1.13021575 0.16270384 2.45447300
[13,] 0.60596072 1.4225296 3.443596530 0.54847576 0.83395115 0.34840708 0.74702202
[14,] 0.00000000 0.2469229 0.568875172 0.97689483 0.02607481 0.33941611 0.08000618
[15,] 0.24692289 0.0000000 0.238242958 1.20594171 0.29994163 0.53558485 0.50468966
[16,] 0.56887517 0.2382430 0.000000000 0.07726078 0.37931103 0.42332406 0.26344865
[17,] 0.97689483 1.2059417 0.077260784 0.00000000 0.25828365 1.07024705 0.61204736
[18,] 0.02607481 0.2999416 0.379311034 0.25828365 0.00000000 0.47533398 0.71325055
[19,] 0.33941611 0.5355848 0.423324059 1.07024705 0.47533398 0.00000000 0.08996771
[20,] 0.08000618 0.5046897 0.263448650 0.61204736 0.71325055 0.08996771 0.00000000
```

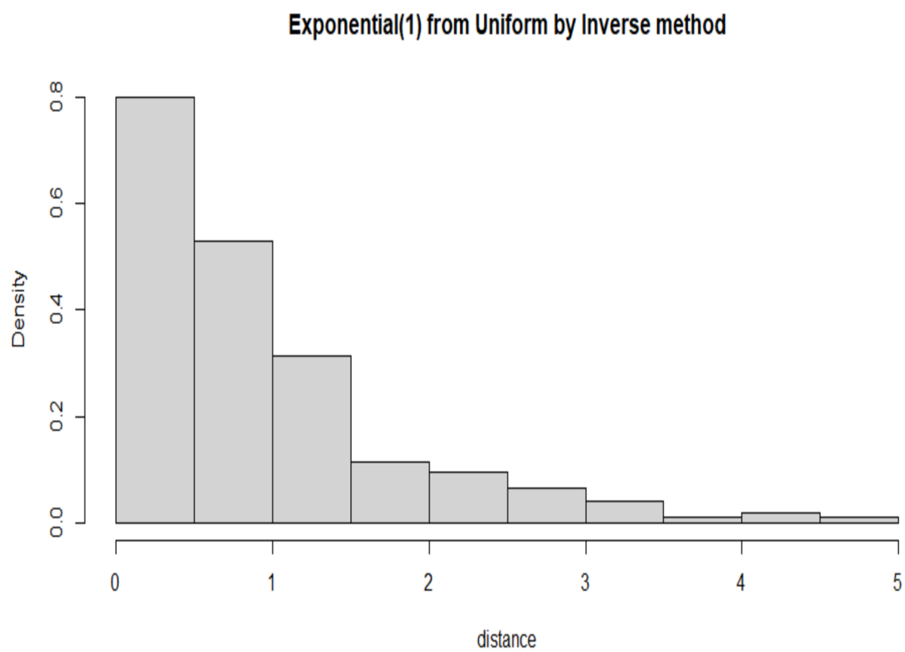

Figure 2.1: Distribution of distances between cities follows Exponential(1) distribution

(4) Choose some large n and m and use Metropolis to sample uniformly from the circuits, and compute the average travel time (so f applied to a circuit is the travel time).

To minimize code size as well as to get an utterly efficient algorithm, user-defined functions are used. In particular, functions FindValidRoute(), SwapRoute(), and CheckDuplicate() are used to simulate the initial state X_0, to interchange position for next state, and to examine repetition throughout the run, respectively.

The two parameters m and n denote the number of runs needed to break in equilibrium and the number of iterations taken into the calculation of the average time spent in each circuit when the chain is in equilibrium. The chain is run for a long period of time (m + n iterations). When the chain is in equilibrium (after the first m iterations), the duplicated chains are eliminated to avoid the biased result.

In the first run, the two parameters are m = 10,000 and n = 10,000. The first valid state X_0 = (20, 8, 15, 13, 18, 6, 5, 1, 9, 7, 19, 12, 16, 14, 11, 10, 2, 3, 17, 4) is chosen uniformly randomly. The average time spent in the first m iterations is 16.39549, which is approximately equal to that of the rest of n iterations (16.46316) when the chain is assumed to have reached equilibrium already. It took about 1 minute for first run.

```
> # Generate first circuit
```

```
> FindValidRoute <- function(N, distance){
+   saveNodes = NULL; routeNodes = NULL;
+   while(isTRUE(1==1)){
+     #cat("count:");  print(count);
+     saveNodeCount = 0;  routeNodeCount = 0;
+     currentNode <- rdunif(1, 1, N)
+     routeNodeCount = routeNodeCount + 1;
+     routeNodes[routeNodeCount] = currentNode
+     saveNodeCount = saveNodeCount + 1;
+     saveNodes[currentNode] <- currentNode;
+     #cat("saveNodes:");  print(saveNodes);
+     #cat("routeNodes:");  print(routeNodes);
+     while(routeNodeCount < N && saveNodeCount < N){
+       proposalNode <- rdunif(1, 1, N)
+       if(is.na(saveNodes[proposalNode])==FALSE) next
+       saveNodeCount = saveNodeCount + 1; saveNodes[proposalNode] <- proposalNode
+       if(isTRUE(distance[currentNode, proposalNode] <= 0)) next
+       routeNodeCount = routeNodeCount + 1;
+       routeNodes[routeNodeCount] <- proposalNode
+       currentNode <- proposalNode
+       #cat("saveNodes:");  print(saveNodes);
+       #cat("routeNodes:");  print(routeNodes);
+     }
+     # print(routeNodes)
+     if(isTRUE(routeNodeCount==N) && isTRUE(distance[routeNodes[1], routeNodes[N]] > 0)){
+       cat("Valid route: ");print(routeNodes)
+       return(routeNodes)
+     }
+     else{
+       cat("Invalid route: ");print(routeNodes)
+     }
+     routeNodes = NULL
+     saveNodes = NULL
+   }
+ }
> SwapRoute <- function(routes, interchangePos1, interchangePos2){
+   newRoutes = routes;
+   newRoutes[interchangePos1] = routes[interchangePos2]
+   newRoutes[interchangePos2] = routes[interchangePos1]
+   return(newRoutes)
+ }
> CheckDuplicate <- function(saveList, checkList){
+
+   if (is.null(saveList))  return(FALSE)
+
+   for(idx1 in 1:length(saveList)){
+     subList =  saveList[[idx1]]
+
+     count = 0;
+     len = length(subList)
+     for(idx2 in 1:len){
+       if (is.null(subList[[idx2]])) next
+       if (is.null(checkList[idx2])) next
+       if ( subList[[idx2]]!=checkList[[idx2]]) break;
+       count = count + 1
+     }
+     if (count==len)  return(TRUE)
+   }
+   return(FALSE)
+ }
> AddToList <- function(saveList,checkList){
+
+   len = length(saveList) + 1
+   saveList[[len]] <- checkList
+   return(saveList)
+ }
>
```

```
> curProposalState = FindValidRoute(N, distance)
Valid route:  [1] 20  8 15 13 18  6  5  1  9  7 19 12 16 14 11 10  2  3 17  4
> totalDistanceM = 0; totalDistanceN = 0;
> totalDuplication = 0
> saveDuplication = NULL
>
> for(runIdx in 1:(m+n)){
+ #for(runIdx in 1:1){
+   # print(runIdx)
+   # Ensure interchangePos1 is less than interchangePos2
+   interchangePos1 = rdunif(1, 1, N)
+   interchangePos2 = rdunif(1, 1, N)
+   if (interchangePos1>interchangePos2){
+     tmp = interchangePos2
+     interchangePos2 =  interchangePos1
+     interchangePos1 =  tmp
+   }
+   #cat("interchangePos ", interchangePos1, interchangePos2); print("")
+
+   # Swap route
+   newProposalState = SwapRoute(curProposalState,interchangePos1,interchangePos2);
+   list1 = NULL
+   if (interchangePos1==1) list1 = list(newProposalState[1],newProposalState[2])
+   else list1 = list(newProposalState[interchangePos1-1], newProposalState[interchangePos1], newProposalState[interchangePos1+1])
+   if (DistanceIsValid(list1, distance)==FALSE) next
+
+   list2 = NULL
+   if (interchangePos2==N) list2 = list(newProposalState[N-1],newProposalState[N])
+   else list2 = list(newProposalState[interchangePos2-1], newProposalState[interchangePos2], newProposalState[interchangePos2+1])
+   if (DistanceIsValid(list2, distance)==FALSE) next
+
+   list3 = list(newProposalState[1], newProposalState[N])
+   if (DistanceIsValid(list3, distance)==FALSE) next
+
+   # Check duplication
+   fDuplication = FALSE
+   if (runIdx>m){
+       fDuplication = CheckDuplicate(saveDuplication, newProposalState)
+       if (fDuplication){
+           totalDuplication = totalDuplication + 1
+       }
+       else {
+           saveDuplication = AddToList(saveDuplication, newProposalState)
+       }
+   }
+   totalDistance = 0;
+   if (isFALSE(fDuplication)){
+       totalDistance = GetTotalDistance(newProposalState, distance)
+   }
+
+   if (runIdx>m) {
+       totalDistanceN = totalDistanceN + totalDistance
+   }
+   else{
+     totalDistanceM = totalDistanceM + totalDistance
+   }
+
+   # cat("newProposalState"); print(newProposalState)
+   curProposalState = newProposalState
+ }
>
> avgDistanceM = totalDistanceM/m
>
> if (n>totalDuplication){
+   avgDistanceN = totalDistanceN/(n-totalDuplication)
```

```
+ } else { avgDistanceN = GetTotalDistance(curProposalState, distance)
+ }
>
> cat("Total duplication M"); print(totalDuplication)
Total duplication M[1] 531
> cat("Total distance M"); print(totalDistanceM)
Total distance M[1] 163954.9
> cat("Total distance N"); print(totalDistanceN)
Total distance N[1] 155889.6
> cat("Average distance M"); print(avgDistanceM)
Average distance M[1] 16.39549
> cat("Average distance N"); print(avgDistanceN)
Average distance N[1] 16.46316
```

(5) Check if your choice of n and m is robust enough, by trying this with 2n and 2m instead, and seeing if your answers change much. If they do, try again with larger values of n or m. Make sure you use the same travel distances as before The product is supposed to be very nice and cool. It shall satisfy all conditions and look awesome.

In the second run, the two parameters are m = 20,000 and n = 20,000. The first valid state X_0 = (19, 3, 2, 20, 17, 14, 7, 18, 16, 12, 10, 13, 8, 11, 9, 4, 1, 15, 5, 6) is chosen uniformly randomly. The average time spent in the first m iterations is 16.64739, which is approximately equal to that of the rest n iterations (16.40367). It took roughly 3 minutes for second run.

```
> m = 20000 # Number of circuits needed to break in equilibrium
> n = 20000 # Number of circuits taken into the calculation of average time spent.
> M = n + m # Total number of runs
> SwapRoute <- function(routes, interchangePos1, interchangePos2){
+   newRoutes = routes;
+   newRoutes[interchangePos1] = routes[interchangePos2]
+   newRoutes[interchangePos2] = routes[interchangePos1]
+   return(newRoutes)
+ }
> CheckDuplicate <- function(saveList, checkList){
+
+   if (is.null(saveList))  return(FALSE)
+
+   for(idx1 in 1:length(saveList)){
+     subList =  saveList[[idx1]]
+
+     count = 0;
+     len = length(subList)
+     for(idx2 in 1:len){
+       if (is.null(subList[[idx2]])) next
+       if (is.null(checkList[idx2])) next
+       if ( subList[[idx2]]!=checkList[[idx2]]) break;
+       count = count + 1
+     }
+     if (count==len)  return(TRUE)
+   }
+   return(FALSE)
+ }
> AddToList <- function(saveList,checkList){
+
+   len = length(saveList) + 1
+   saveList[[len]] <- checkList
+   return(saveList)
+ }
>
> curProposalState = FindValidRoute(N, distance)
Invalid route:  [1] 11  9  5 13 19  4 12 10 20 15  3 16 14 17  7  8  1  2 18
Valid route:  [1] 19  3  2 20 17 14  7 18 16 12 10 13  8 11  9  4  1 15  5  6
> totalDistanceM = 0; totalDistanceN = 0;
> totalDuplication = 0
```

```
> saveDuplication = NULL
>
> for(runIdx in 1:(m+n)){
+ #for(runIdx in 1:1){
+   # print(runIdx)
+   # Ensure interchangePos1 is less than interchangePos2
+   interchangePos1 = rdunif(1, 1, N)
+   interchangePos2 = rdunif(1, 1, N)
+   if (interchangePos1>interchangePos2){
+     tmp = interchangePos2
+     interchangePos2 =  interchangePos1
+     interchangePos1 =  tmp
+   }
+   #cat("interchangePos ", interchangePos1, interchangePos2); print("")
+
+   # Swap route
+   newProposalState = SwapRoute(curProposalState,interchangePos1,interchangePos2);
+   list1 = NULL
+   if (interchangePos1==1) list1 = list(newProposalState[1],newProposalState[2])
+   else list1 = list(newProposalState[interchangePos1-1], newProposalState[interchangePos1], newProposalState[interchangePos1+1])
+   if (DistanceIsValid(list1, distance)==FALSE) next
+
+   list2 = NULL
+   if (interchangePos2==N) list2 = list(newProposalState[N-1],newProposalState[N])
+   else list2 = list(newProposalState[interchangePos2-1], newProposalState[interchangePos2], newProposalState[interchangePos2+1])
+   if (DistanceIsValid(list2, distance)==FALSE) next
+
+   list3 = list(newProposalState[1], newProposalState[N])
+   if (DistanceIsValid(list3, distance)==FALSE) next
+
+   # Check duplication
+   fDuplication = FALSE
+   if (runIdx>m){
+       fDuplication = CheckDuplicate(saveDuplication, newProposalState)
+       if (fDuplication){
+           totalDuplication = totalDuplication + 1
+       }
+       else {
+           saveDuplication = AddToList(saveDuplication, newProposalState)
+       }
+   }
+   totalDistance = 0;
+   if (isFALSE(fDuplication)){
+       totalDistance = GetTotalDistance(newProposalState, distance)
+   }
+
+   if (runIdx>m) {
+       totalDistanceN = totalDistanceN + totalDistance
+   }
+   else{
+     totalDistanceM = totalDistanceM + totalDistance
+   }
+
+   # cat("newProposalState"); print(newProposalState)
+   curProposalState = newProposalState
+ }
>
> avgDistanceM = totalDistanceM/m
>
> if (n>totalDuplication){
+   avgDistanceN = totalDistanceN/(n-totalDuplication)
+ } else { avgDistanceN = GetTotalDistance(curProposalState, distance)
+ }
>
> cat("Total duplication M"); print(totalDuplication)
```

```
Total duplication M[1] 1118
> cat("Total distance M"); print(totalDistanceM)
Total distance M[1] 332947.7
> cat("Total distance N"); print(totalDistanceN)
Total distance N[1] 309734.1
> cat("Average distance M"); print(avgDistanceM)
Average distance M[1] 16.64739
> cat("Average distance N"); print(avgDistanceN)
Average distance N[1] 16.40367
```

Summary: As the two average times of the first m iterations in two runs are approximately equal (16.3954 when m = 10,000 and 16.64739 when m = 20,000), the initial choice of n and m is sufficiently robust, and we can claim that the chain is in equilibrium after m runs. It also indicates that the chain is stationary as the output value does not depend on the initial state X_0. This means that for every k, the distribution of $(X_1; ...; X_k)$ is the same as that of $(X_{t+1}; ...; X_{t+k})$ for every t.

b Check that this Q is symmetric

In the Traveling Salesman Problem, each state $\vec{X}_n = (a_1, a_2, ..., a_{20})$ is a circuit of 20 cities. As above, each state is numbered from 1 to 20!

The proposal transition matrix Q is of size 20! $\times$ 20!. Each entry q_{ij} in the Q matrix represents the rate that the chain moves from state i to state j where i and j $\in$ S.

By the algorithm above, the chain moves from one state to another state or stay put by interchanging the position of two randomly uniformly chosen cities with probability $\frac{1}{\binom{20}{2}} = \frac{1}{190}$. The chain stays put when there is at least one forbidden pair in the route proposed by Q.

$$Q = \begin{vmatrix} -q_1 & q_{1,2} & q_{1,3} & \cdots & q_{1,20!} \\ q_{2,1} & -q_2 & q_{2,3} & \cdots & q_{2,20!} \\ q_{3,1} & q_{3,2} & -q_3 & \cdots & q_{3,20!} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ q_{20!,1} & q_{20!,2} & q_{20!,3} & \cdots & -q_{20!} \end{vmatrix}$$

where the entries on the diagonal are defined as $q_i = \sum_{k \neq i} q_{ik}$

- If the transition from state i to state j by the interchange of two random cities creates at least one disconnected pair of cities, then $q_{ij} = 0$. In follows that j is an invalid route and thus $q_{ji} = 0$.
- If the transition from state i to state j by the interchange of two random cities creates no disconnected pair of cities, then $q_{ij} = \frac{1}{190}$. It follows that state j is a valid list of 20 cities, and it will only return to state the list of 20 cities of state i with rate $q_{ji} = \frac{1}{190}$

Therefore, Q is symmetric.

3. R Code Appendix:

```r
set.seed(12345)
# Packages used
install.packages("combinat")
library(combinat)
install.packages("purrr")
library(purrr)
# Variable declaration
N = 20     # Number of cities
p = 5      # Number of disconnected pairs
m = 10000 # Number of circuits needed to break in equilibrium
n = 10000 # Number of circuits taken into the calculation of average time spent.
M = n + m # Total number of runs
lamda = 1 # Exponential parameter
x = 0; y = 0; z = 0;
# (1) Label 20 cities as 1, 2, . . . , 20
cities <- c(1:N)
cities
# (2) Pick 5 pairs of cities {i, j} for which direct travel between i and j is forbidden
# Number of pairs of cities
pairs <- dim(combn(N, 2))[2]
pairs
for(a in 1:p){
  x[a] = rdunif(1, 1, N);
  y[a] = rdunif(1, 1, N)
  for(b in 1:(a-1)){
    if(isTRUE(x[a] != y[a] || y[a] != y[a - b] && x[a] != x[a - b])){a = a + 1}
    else{a = a + 0}
  }
}
disconnected.pairs <- data.frame(x, y)
disconnected.pairs #Random disconnected pairs
# (3) Generate random positive distances between the other 185 pairs of cities
unif.matrix <- matrix(runif(N*N), N)          # Initiate U(0,1) random variables
distance = - (1/lamda)*log(unif.matrix)       # Inverse method - Sampling from Exp(1)
ind <- lower.tri(distance)
distance[ind] <- t(distance)[ind]             # d(i,j) = d(j,i)
hist(distance,freq=F,main="Exponential(1) from Uniform by Inverse method")
for(i in 1:N){
  for(j in 1:N){
    for(k in 1:p){
      if(isTRUE(i == j)){distance[i, j] = 0} # d(i,i) = 0
      else if(isTRUE(distance[i, j] == distance[disconnected.pairs$x[k], disconnected.pairs$y[k]])){
          distance[i, j] = 0} #distance of disconnected pairs is 0
      else if(isTRUE(distance[i, j] == distance[disconnected.pairs$y[k], disconnected.pairs$x[k]])){
          distance[i, j] = 0}
      else{distance[i, j] = distance[i, j]}
    }
  }
}
distance
DistanceIsValid <- function(lst, distance){
  len = length(lst);
  if (len <2) return(FALSE)
```

```
  if (isTRUE(distance[lst[[1]], lst[[len]]]<=0)){
    return(FALSE)
  }
  for(idx in 2:len){
    if (isTRUE(distance[lst[[idx-1]], lst[[idx]]]<=0)){
      return(FALSE)
    }
  }
  return(TRUE)
}
GetTotalDistance <- function(list, distance){

    len = length(list)
    if (len < 2) return (0)
    totalDistance = totalDistance + distance[list[1], list[len]]
    for(idx in 2:len){
        totalDistance = totalDistance + distance[list[idx-1], list[idx]]
    }
    return(totalDistance)
}
# (4) Sample uniformly from the circuits, and compute the average travel time
# Generate first circuit
FindValidRoute <- function(N, distance){
  saveNodes = NULL; routeNodes = NULL;
  while(isTRUE(1==1)){
    #cat("count:");  print(count);
    saveNodeCount = 0;  routeNodeCount = 0;
    currentNode <- rdunif(1, 1, N)
    routeNodeCount = routeNodeCount + 1;
    routeNodes[routeNodeCount] = currentNode
    saveNodeCount = saveNodeCount + 1;
    saveNodes[currentNode] <- currentNode;
    #cat("saveNodes:");  print(saveNodes);
    #cat("routeNodes:");  print(routeNodes);
    while(routeNodeCount < N && saveNodeCount < N){
      proposalNode <- rdunif(1, 1, N)
      if(is.na(saveNodes[proposalNode])==FALSE) next
      saveNodeCount = saveNodeCount + 1; saveNodes[proposalNode] <- proposalNode
      if(isTRUE(distance[currentNode, proposalNode] <= 0)) next
      routeNodeCount = routeNodeCount + 1;
      routeNodes[routeNodeCount] <- proposalNode
      currentNode <- proposalNode
      #cat("saveNodes:");  print(saveNodes);
      #cat("routeNodes:");  print(routeNodes);
    }
    # print(routeNodes)
    if(isTRUE(routeNodeCount==N) && isTRUE(distance[routeNodes[1], routeNodes[N]] > 0)){
      cat("Valid route: ");print(routeNodes)
      return(routeNodes)
    }
    else{
      cat("Invalid route: ");print(routeNodes)
    }
    routeNodes = NULL
    saveNodes = NULL
  }
}
SwapRoute <- function(routes, interchangePos1, interchangePos2){
  newRoutes = routes;
  newRoutes[interchangePos1] = routes[interchangePos2]
  newRoutes[interchangePos2] = routes[interchangePos1]
  return(newRoutes)
}
CheckDuplicate <- function(saveList, checkList){

  if (is.null(saveList))  return(FALSE)

```

```
  for(idx1 in 1:length(saveList)){
    subList =  saveList[[idx1]]

    count = 0;
    len = length(subList)
    for(idx2 in 1:len){
      if (is.null(subList[[idx2]])) next
      if (is.null(checkList[idx2])) next
      if ( subList[[idx2]]!=checkList[[idx2]]) break;
      count = count + 1
    }
    if (count==len)  return(TRUE)
  }
  return(FALSE)
}
AddToList <- function(saveList,checkList){

  len = length(saveList) + 1
  saveList[[len]] <- checkList
  return(saveList)
}

curProposalState = FindValidRoute(N, distance)
totalDistanceM = 0; totalDistanceN = 0;
totalDuplication = 0
saveDuplication = NULL

for(runIdx in 1:(m+n)){
#for(runIdx in 1:1){
  # print(runIdx)
  # Ensure interchangePos1 is less than interchangePos2
  interchangePos1 = rdunif(1, 1, N)
  interchangePos2 = rdunif(1, 1, N)
  if (interchangePos1>interchangePos2){
    tmp = interchangePos2
    interchangePos2 =  interchangePos1
    interchangePos1 =  tmp
  }
  #cat("interchangePos ", interchangePos1, interchangePos2); print("")

  # Swap route
  newProposalState = SwapRoute(curProposalState,interchangePos1,interchangePos2);
  list1 = NULL
  if (interchangePos1==1) list1 = list(newProposalState[1],newProposalState[2])
  else list1 = list(newProposalState[interchangePos1-1], newProposalState[interchangePos1],
      newProposalState[interchangePos1+1])
  if (DistanceIsValid(list1, distance)==FALSE) next

  list2 = NULL
  if (interchangePos2==N) list2 = list(newProposalState[N-1],newProposalState[N])
  else list2 = list(newProposalState[interchangePos2-1], newProposalState[interchangePos2],
      newProposalState[interchangePos2+1])
  if (DistanceIsValid(list2, distance)==FALSE) next

  list3 = list(newProposalState[1], newProposalState[N])
  if (DistanceIsValid(list3, distance)==FALSE) next

  # Check duplication
  fDuplication = FALSE
  if (runIdx>m){
      fDuplication = CheckDuplicate(saveDuplication, newProposalState)
      if (fDuplication){
          totalDuplication = totalDuplication + 1
      }
      else {
          saveDuplication = AddToList(saveDuplication, newProposalState)
      }
```

```
  }
  totalDistance = 0;
  if (isFALSE(fDuplication)){
      totalDistance = GetTotalDistance(newProposalState, distance)
  }

  if (runIdx>m) {
      totalDistanceN = totalDistanceN + totalDistance
  }
  else{
    totalDistanceM = totalDistanceM + totalDistance
  }

  # cat("newProposalState"); print(newProposalState)
  curProposalState = newProposalState
}

avgDistanceM = totalDistanceM/m

if (n>totalDuplication){
  avgDistanceN = totalDistanceN/(n-totalDuplication)
} else { avgDistanceN = GetTotalDistance(curProposalState, distance)
}

cat("Total duplication M"); print(totalDuplication)
cat("Total distance M"); print(totalDistanceM)
cat("Total distance N"); print(totalDistanceN)
cat("Average distance M"); print(avgDistanceM)
cat("Average distance N"); print(avgDistanceN)
```

4. References

Madras. N. (2001) *Lectures on Monte Carlo Methods.* AMS and Fields Institute.

Peskun. P. (2020). MATH 4931. *Simulation and the Monte Carlo Methods. Inverse C.D.F method.* York University.

Robert.C.P & Casella.G.(2010). *Introducing Monte Carlo Methods with R.* Springer.