Stat 102C HW4: Answer Key

Jonathan Arfa

June 2, 2014

Problem 1.1

$$p^{(t+1)}(y) = \sum_x p^{(t)}(x)K(x,y) \tag{1}$$

$$p(X_{t+1} = y) = \sum_x p(X_{t+1} = y \,\&\, X_t = x) \tag{2}$$

$$= \sum_x p(X_{t+1} = y | X_t = x)\, p(X_t = x) \tag{3}$$

$$= \sum_x K(x,y)\, p^{(t)}(x) \tag{4}$$

$$\tag{5}$$

Problem 1.2

$$p^{(t)} = p^{(t-1)}K \tag{6}$$

$$= p^{(t-2)}KK \tag{7}$$

$$= p^{(0)}K^t \tag{8}$$

Problem 1.3

Let's say that $s = 2$

$$K^{(2)}(x,y) = p(X_{t+2} = y | X_t = x) \tag{9}$$

$$= \sum_z p(X_{t+2} = y \,\&\, X_{t+1} = z | X_t = x) \tag{10}$$

$$= \sum_z p(X_{t+2} = y | X_{t+1} = z \,\&\, X_t = x)\, p(X_{t+1} = z | X_t = x) \tag{11}$$

$$= \sum_z K(x,z)K(z,y) = [K * K](x,y) \tag{12}$$

$$K^{(2)}(x,y) = K^2(x,y) \tag{13}$$

Problem 1.4

In 1.1, suppose a population of 1 million people is moving around all the states. $p^{(t+1)}(y)$ can be interpreted as the number of people in state y at time $t+1$, and $p^{(t)}(x)$ can be interpreted as the number of people in state x at time t. $K(x,y)$ can be interpreted as, among all the people in state x at time t, the fraction that will move to y at time $t+1$. So $p^{(t)}(x)K(x,y)$ is the number of people who are at x at time t and who are at y at time $t+1$. So $p^{(t+1)}(y)$ should be the sum of $p^{(t)}(x)K(x,y)$ over all states x.

In 1.2, the distribution of the population at time t is $p^{(t)}$. This distribution evolves over time and will eventually approach the stationary distribution.

In 1.3, The transition matrix for time s, $K^{(s)}(x, y)$ can be intrepreted as, among all the people in state x at time t, the fraction that will end up in state y at time $t + s$.

Problem 2

$$\pi(x)K(x,y) = \pi(y)K(y,x) \tag{14}$$
$$\sum_x \pi(x)K(x,y) = \sum_x \pi(y)K(y,x) \tag{15}$$
$$\sum_x \pi(x)K(x,y) = \pi(y)\sum_x K(y,x) \tag{16}$$
$$\sum_x \pi(x)K(x,y) = \pi(y) \tag{17}$$

Note that $\sum_x K(y, x) = 1$.

Interpretation of $\sum_x \pi(x)K(x, y) = \pi(y)$: this is the property of stationary distribution. The distribution of the population does not change once it is at stationarity, meaning that the number of people in each state does not change, even though people are still moving around.

Intrpretation of $\pi(x)K(x, y) = \pi(y)K(y, x)$: this is detailed balance, meaning that for every pair of states x, y, the number of people who go from x to y is the same as the number of people who go from y to x. If this is true for every pair of states, then the number of people in each state does not change.

Problem 3.1

"Suppose the base chain is such that at each state, we move to one of the other two states with probability 1/2."

```
p = c(0.4, 0.2, 0.4)
B = matrix(c(0, 1/2, 1/2, 1/2, 0, 1/2, 1/2, 1/2, 0), byrow = TRUE, nrow = 3)
print(B)
```

```
##      [,1] [,2] [,3]
## [1,]  0.0  0.5  0.5
## [2,]  0.5  0.0  0.5
## [3,]  0.5  0.5  0.0
```

Problem 3.2

This following code is far more verbose and slow than it needs to be, I'm just trying to be very clear on what's happening

```
Acceptance = matrix(nrow = 3, ncol = 3)
for (x in 1:3) {
    for (y in 1:3) Acceptance[x, y] = min(1, p[y]/p[x])  #this is min(1, f(y)/f(x))
    # qxy and qyx are unnecessary since qxy == qyx
}
Acceptance
```

```
##      [,1] [,2] [,3]
## [1,]    1  0.5    1
## [2,]    1  1.0    1
## [3,]    1  0.5    1
```

```
(moveMatrix = B * Acceptance)  #note that this is an element-wise product
```

```
##      [,1] [,2] [,3]
## [1,]  0.0 0.25  0.5
## [2,]  0.5 0.00  0.5
## [3,]  0.5 0.25  0.0
```

```
rowSums(moveMatrix)
```

```
## [1] 0.75 1.00 0.75
```

```
# note that the rows of moveMatrix don't sum to 1.
M = moveMatrix
for (i in 1:3) {
    # the prob of staying in one place is 1-prob(moving)
    M[i, i] = 1 - sum(M[i, -i])
}
M
```

```
##      [,1] [,2] [,3]
## [1,] 0.25 0.25 0.50
## [2,] 0.50 0.00 0.50
## [3,] 0.50 0.25 0.25
```

```
rowSums(M)
```

```
## [1] 1 1 1
```

Problem 3.3

There are two ways to do show that it converges. The simple way, or the way we did it in Homework #3.

```
matpow = function(mat, pow) {
    # this function multiplies a matrix to an exponent. There are packages that
    # have this function, but it's pretty easy to write ourselves.
    stopifnot(pow >= 1, pow%%1 == 0)  #this line just prevents us from giving the function bad input.
    m = mat
    if (pow >= 2)
        for (i in 2:pow) {
            m = m %*% mat
        }
    m
}
M
```

```
##      [,1] [,2] [,3]
## [1,] 0.25 0.25 0.50
## [2,] 0.50 0.00 0.50
## [3,] 0.50 0.25 0.25
```

```
matpow(M, 20)
```

```
##      [,1] [,2] [,3]
## [1,]  0.4  0.2  0.4
## [2,]  0.4  0.2  0.4
## [3,]  0.4  0.2  0.4
```

```
p
```

```
## [1] 0.4 0.2 0.4

# It works!
```

Here's another method using our code from Homework #3.

```
RW3 = function(t = 100, p = c(1, 0, 0), K) {
    # K is our matrix
    stopifnot(all(rowSums(K) == 1))
    states = c(1, 2, 3)
    for (i in 1:t) {
        p1 = sum(p * K[, 1])
        p2 = sum(p * K[, 2])
        p3 = sum(p * K[, 3])
        p = c(p1, p2, p3)
    }
    return(p)
}
RW3(t = 100, p = c(1, 0, 0), M)

## [1] 0.4 0.2 0.4

RW3(t = 100, p = c(0, 1, 0), M)

## [1] 0.4 0.2 0.4

RW3(t = 100, p = c(0, 0, 1), M)

## [1] 0.4 0.2 0.4

p

## [1] 0.4 0.2 0.4

# they're the same
```

Problem 3.4

Again, we're using (modified) code from Homework #3. Check the hw3 solution if you want to see more comments about how this function works.

```
set.seed(5)
RWM = function(t = 100, M = 10000, p0 = 1, K) {
    # note that M inside this function denotes sample size
    stopifnot(all(rowSums(K) == 1))
    states = c(1, 2, 3)
    people = rep(p0, M)
    n = numeric(3)
    index = vector(mode = "list", 3)
    move = vector(mode = "list", 3)
    movements = array(0, dim = c(3, 3, t))
    for (i in 1:t) {
        for (x in 1:3) {
            n[x] = sum(people == x)
            index[[x]] = which(people == x)
            move[[x]] = sample(states, n[x], replace = TRUE, prob = K[x, ])
        }
```

```
        for (x in 1:3) people[index[[x]]] = move[[x]]
    }
    p = c(sum(people == 1), sum(people == 2), sum(people == 3))/M
    p
}
RWM(p0 = 1, K = M)

## [1] 0.3901 0.2012 0.4087

RWM(p0 = 2, K = M)

## [1] 0.4061 0.1961 0.3978

RWM(p0 = 3, K = M)

## [1] 0.4044 0.2068 0.3888
```

All three are converging to the vector π.

Problem 3.5

I've only changed a few lines. Now this function returns two things, the p vector and an array with the matrix of moves for every iteration.

```
set.seed(10)
RWMv2 = function(t = 100, M = 10000, p0 = 1, K) {
    # note that M inside this function denotes sample size
    stopifnot(all(rowSums(K) == 1))
    states = c(1, 2, 3)
    people = rep(p0, M)
    n = numeric(3)
    index = vector(mode = "list", 3)
    move = vector(mode = "list", 3)
    movements = array(0, dim = c(3, 3, t))
    for (i in 1:t) {
        for (x in 1:3) {
            n[x] = sum(people == x)
            index[[x]] = which(people == x)
            move[[x]] = sample(states, n[x], replace = TRUE, prob = K[x, ])
            for (y in 1:3) {
                movements[x, y, i] = sum(move[[x]] == y)
            }
        }
        for (x in 1:3) people[index[[x]]] = move[[x]]
    }
    p = c(sum(people == 1), sum(people == 2), sum(people == 3))/M
    return(list(p = p, M = movements))
}
```

If you specify that the function should return M, it will return a 3x3x100 array, which is way too large. So make sure you specify exactly what you want (like I do below). Let's look at the matrix of movements for some of the last few values of t.

```
RWMv2(p0 = 1, K = M)$M[, , 98:100]

## , , 1
##
##      [,1] [,2] [,3]
## [1,] 1011  963 2104
```

```
## [2,] 1034    0  993
## [3,] 1924 1023  948
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,] 1061  996 1912
## [2,]  998    0  988
## [3,] 2076  982  987
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,] 1083 1065 1987
## [2,] 1021    0  957
## [3,] 1927  993  967
```

Problem 4.1

I ended up writing 150 lines of functions for this problem (including comments). They are attached to the back of this solutions, and will also be available on GitHub.

Change the variable for your file directory below, and you can load the functions into your R workspace with:

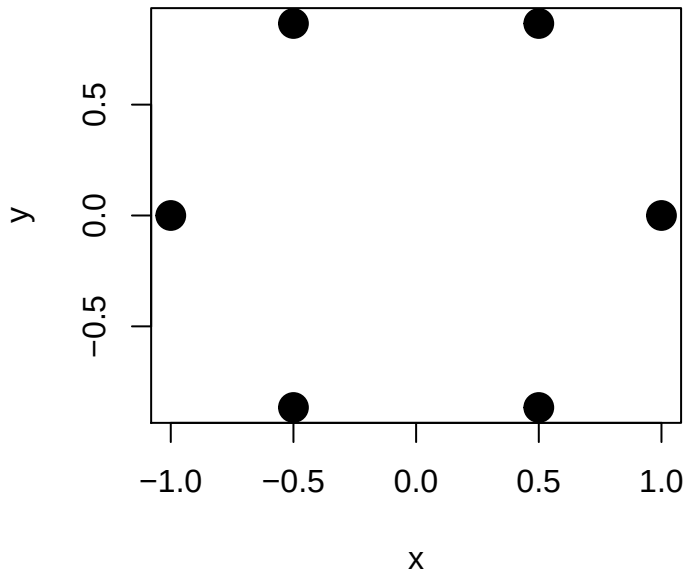
```
file_dir = "/Users/jonathanarfa/Dropbox/Class_Files/TA Stat 102C/HW4/"
source(paste0(file_dir, "hw4-p4-functions.R"))
```

If m=5, then there are 6 cities.

```
m = 5
cities = placeCitiesCircle(m)
# see the back of this document for the code of that function.
print(cities)
```

```
##          x          y
## [1,]  1.0  0.000e+00
## [2,]  0.5  8.660e-01
## [3,] -0.5  8.660e-01
## [4,] -1.0  1.225e-16
## [5,] -0.5 -8.660e-01
## [6,]  0.5 -8.660e-01
```

```
plot(cities, pch = 19, cex = 2)
```




```
# the easy way to find distances: the dist() function
distances = as.matrix(dist(cities))
print(distances)
```

```
##       1     2     3     4     5     6
## 1 0.000 1.000 1.732 2.000 1.732 1.000
## 2 1.000 0.000 1.000 1.732 2.000 1.732
## 3 1.732 1.000 0.000 1.000 1.732 2.000
## 4 2.000 1.732 1.000 0.000 1.000 1.732
## 5 1.732 2.000 1.732 1.000 0.000 1.000
## 6 1.000 1.732 2.000 1.732 1.000 0.000
```

Problem 4.2

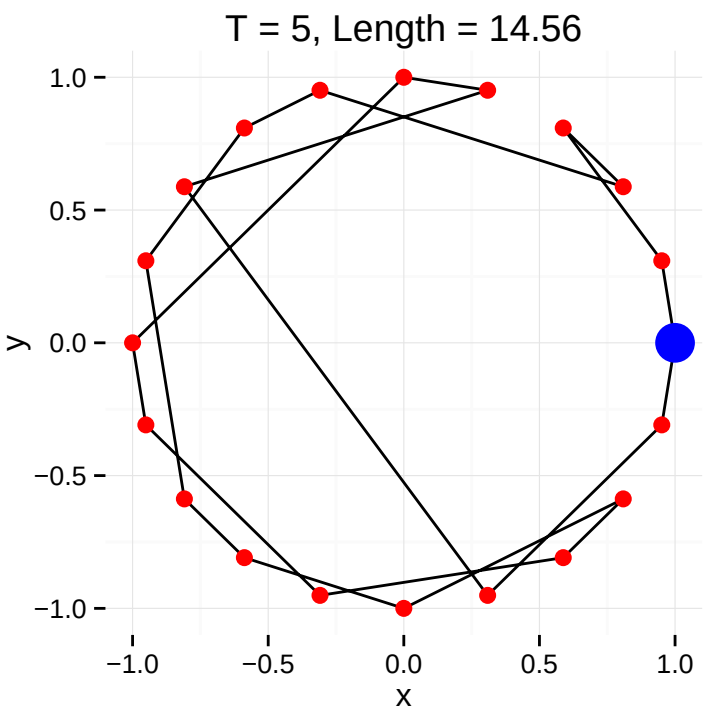
I wrote a function to find the path of U for different parameters (such as Nsim, T, and m). I can also tell the function to automatically plot the shortest path via the parameter plotbest. To save space, the MetropolisWalkCircle() function is only written in the appendix. Check it out.

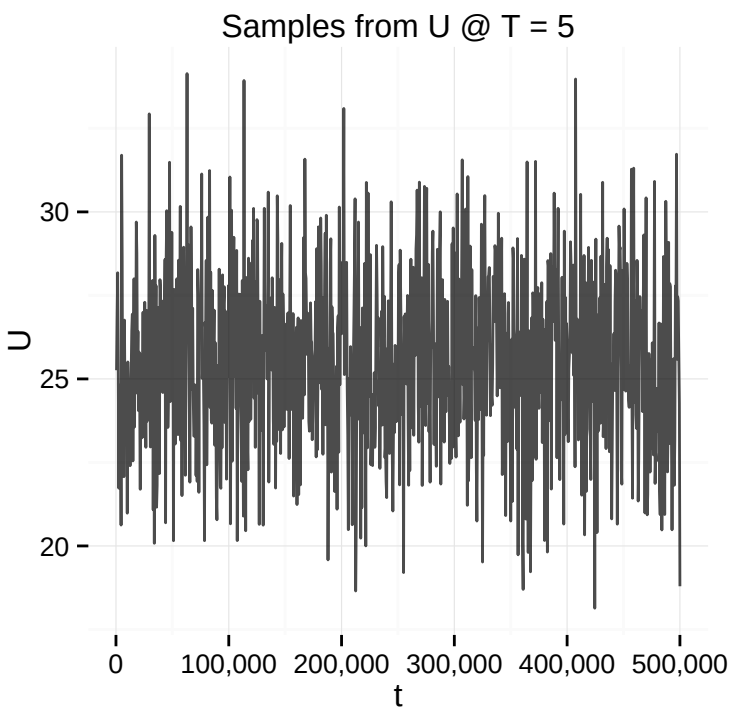
I'm running this sampler at 2 values of T: 5 and 0.2. At each value we'll plot the best path from that it sampled, and the path of U.

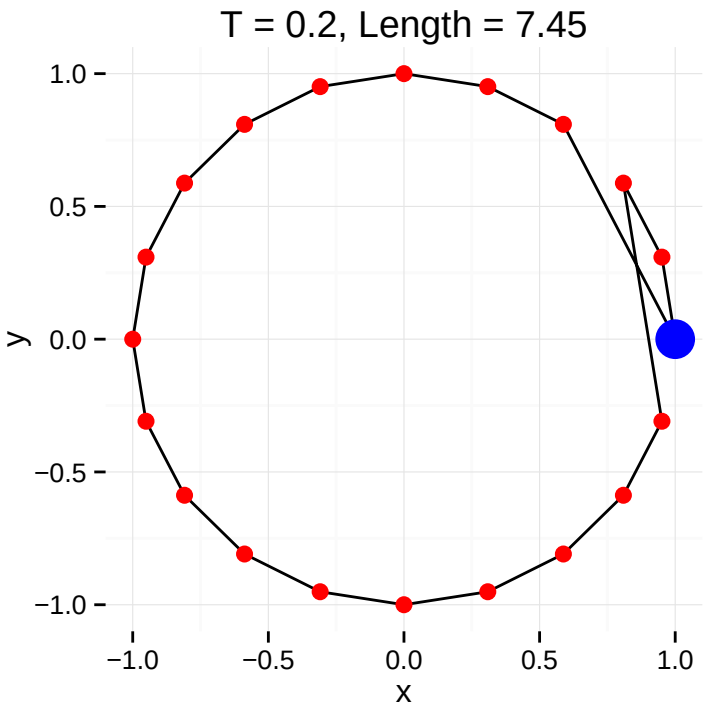
Note that a lower value for T leads to the Markov Chain converging towards lower values of U over time. Also, I had to make the plots only display points where t is a multiple of 500, simply to keep the size of the images down. If you plotted every single point the lines would look even more jagged.

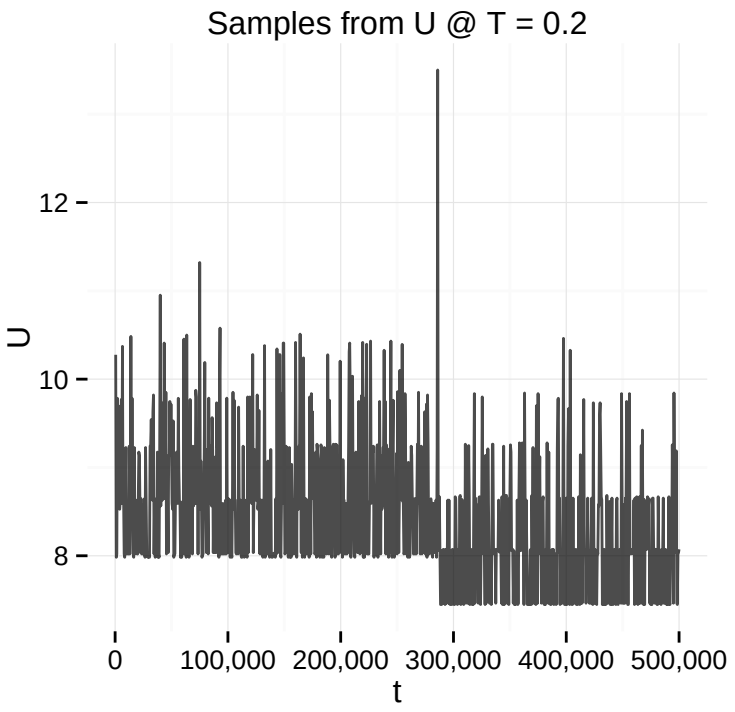
```
library(ggplot2)
library(scales)
mstar = 19
Nsim = 5e+05
set.seed(15)
U = MetropolisWalkCircle(m = mstar, T = 5, Nsim = Nsim, plotbest = TRUE)$U
# The following 2 lines serve to only choose every 100th value of U.
# Plotting every value makes the resulting image too large for the .pdf.
```

```r
U = U[seq_along(U)%%500 == 0]
t = 500 * (1:floor(Nsim/500))  #t is increments of 100
ggplot(data.frame(U, t), aes(x = t, y = U)) + geom_line(alpha = 0.7) + scale_x_continuous(labels = comma) +
    ggtitle(paste("Samples from U @ T =", 5)) + theme_minimal() + theme(plot.title = element_text(size = 12))

set.seed(15)
U = MetropolisWalkCircle(m = mstar, T = 0.2, Nsim = Nsim, plotbest = TRUE)$U
U = U[seq_along(U)%%500 == 0]
t = 500 * (1:floor(Nsim/500))  #t is increments of 100
ggplot(data.frame(U, t), aes(x = t, y = U)) + geom_line(alpha = 0.7) + scale_x_continuous(labels = comma) +
    ggtitle(paste("Samples from U @ T =", 0.2)) + theme_minimal() + theme(plot.title = element_text(size = 12))
```







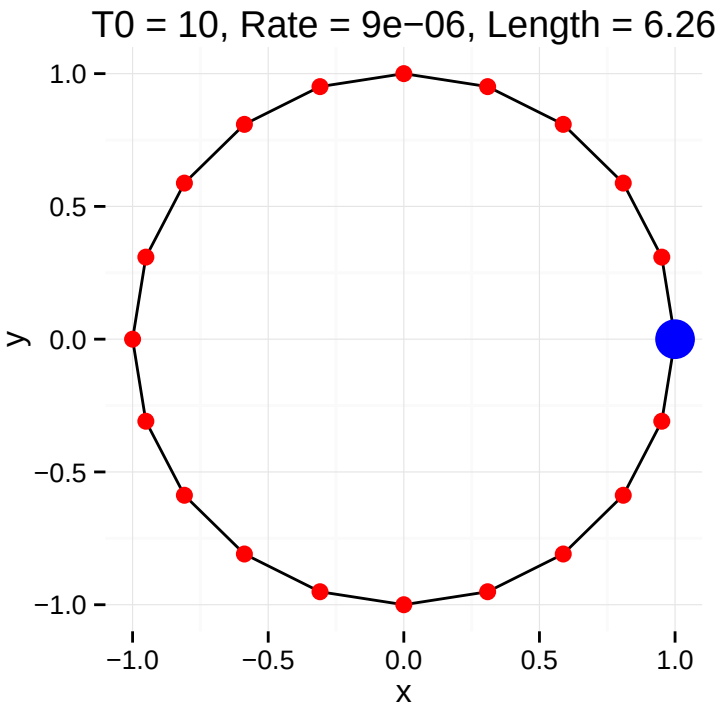


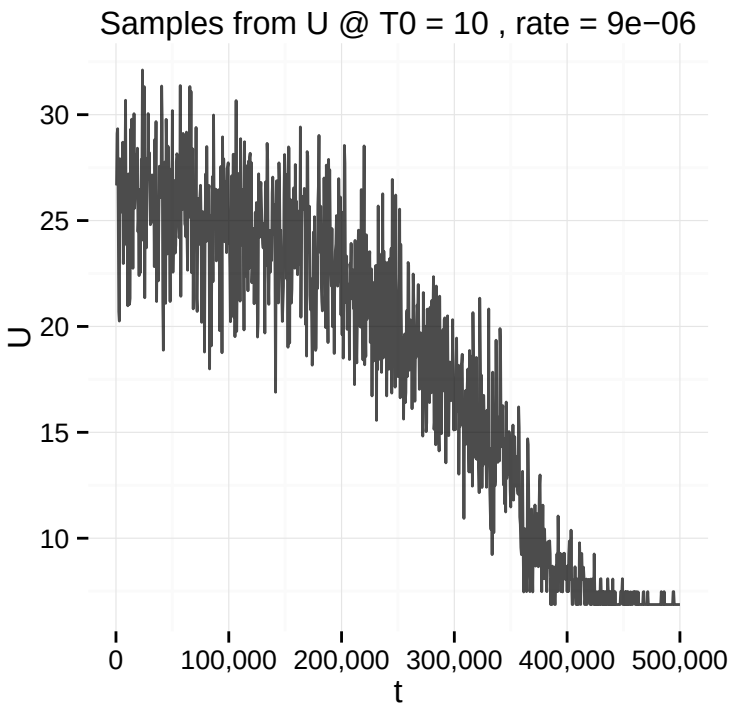
Problem 4.3

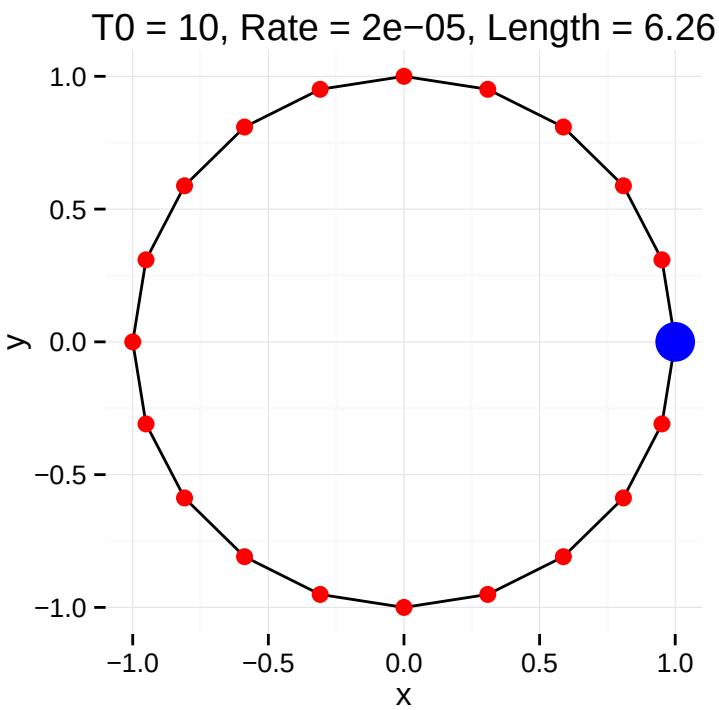
If T_0 is our initial value of T and T_N is the final value of T I want, I can find the cooling rate by $T_N = T_0 * (1 - rate)^{Nsim}$. By solving for $rate$, I get $rate = 1 - (\frac{T_N}{T_0})^{1/Nsim}$

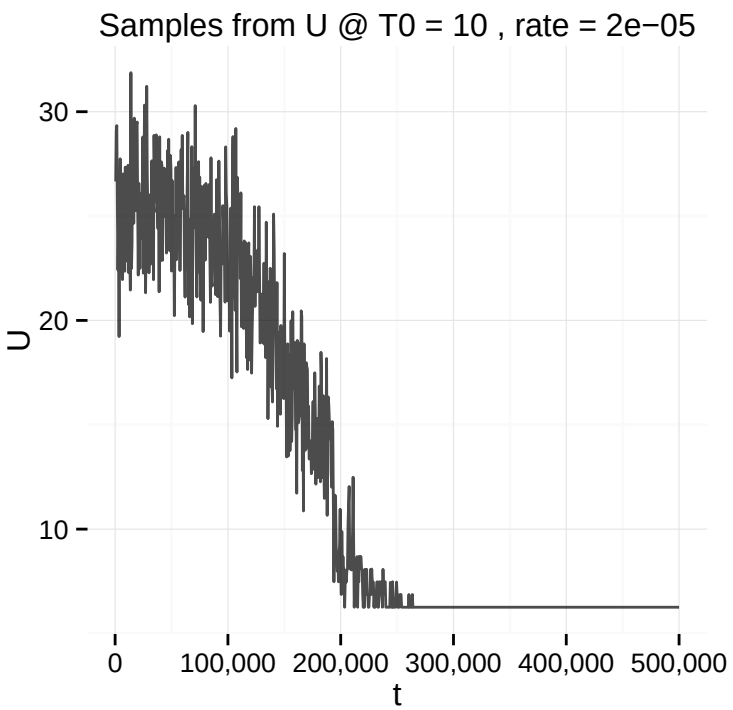
```
library(ggplot2)
library(scales)
mstar = 19
Nsim = 5e+05
T0 = 10
TN = 0.1
coolingrate = 1 - (TN/T0)^(1/Nsim)
print(coolingrate)  #T will be reduced by a factor of 1-coolingrate at each iteration

## [1] 9.21e-06

set.seed(15)
U = SimAnnealWalkCircle(m = mstar, T0 = T0, rate = coolingrate, Nsim = Nsim,
    plotbest = TRUE)$U
U = U[seq_along(U)%%500 == 0]
t = 500 * (1:floor(Nsim/500))
ggplot(data.frame(U, t), aes(x = t, y = U)) + geom_line(alpha = 0.7) + scale_x_continuous(labels = comma) +
    ggtitle(paste("Samples from U @ T0 =", T0, ", rate =", signif(coolingrate,
        1))) + theme_minimal() + theme(plot.title = element_text(size = 12))

T0 = 10
TN = 0.002
(coolingrate = 1 - (TN/T0)^(1/Nsim))

## [1] 1.703e-05
```

```r
set.seed(15)
U = SimAnnealWalkCircle(m = mstar, T0 = T0, rate = coolingrate, Nsim = Nsim,
    plotbest = TRUE)$U
U = U[seq_along(U)%%500 == 0]
t = 500 * (1:floor(Nsim/500))
ggplot(data.frame(U, t), aes(x = t, y = U)) + geom_line(alpha = 0.7) + scale_x_continuous(labels = comma) +
    ggtitle(paste("Samples from U @ T0 =", T0, ", rate =", signif(coolingrate,
        1))) + theme_minimal() + theme(plot.title = element_text(size = 12))
```







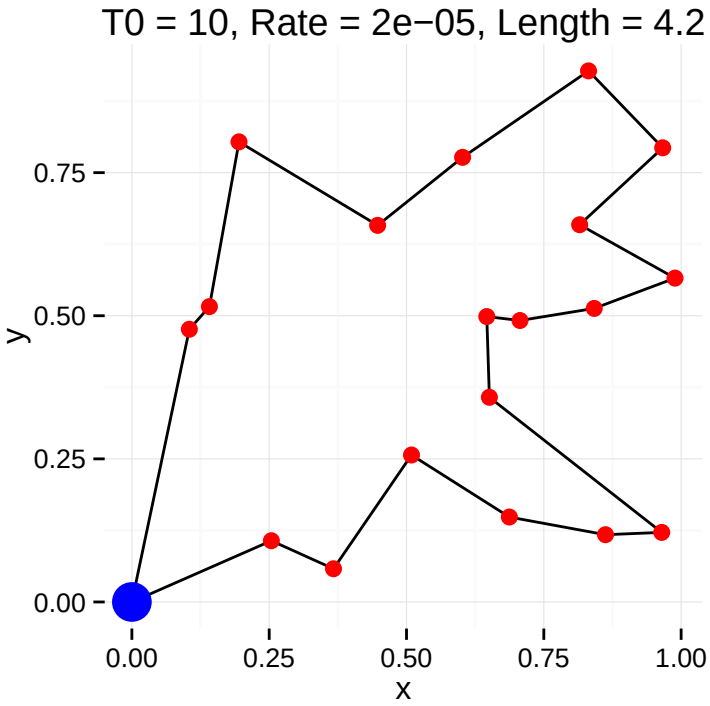


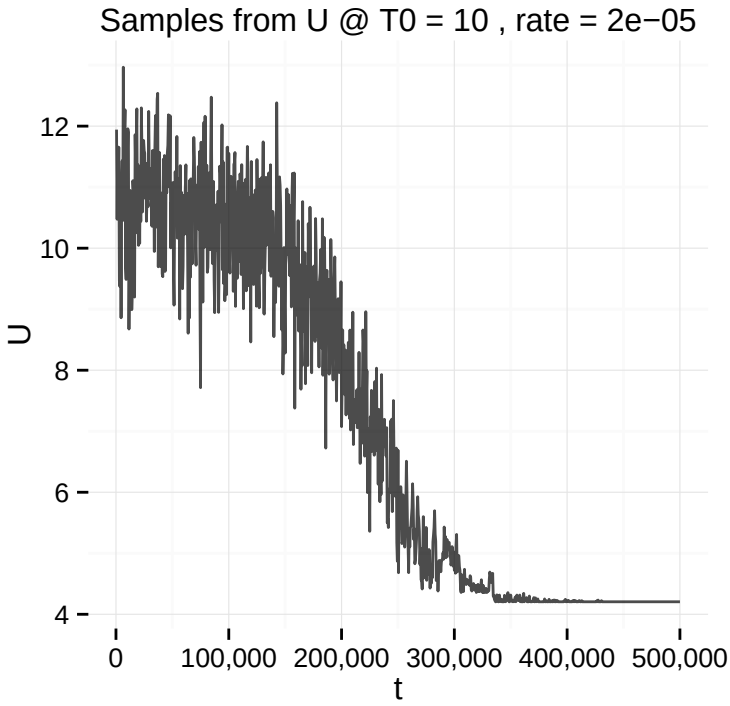
A faster cooling rate made an enormous impact. Remember, I only used 500,000 iterations here, which for MCMC algorithms is not a huge number.

Problem 4.4

This function is pretty similar, aside from the fact that it places the cities in a different manner. See the placeCitiesSquare() in the appendix.

```
set.seed(15)
Nsim = 5e+05
mstar = 19
T0 = 10
TN = 0.002
(coolingrate = 1 - ((TN/T0)^(1/Nsim)))

## [1] 1.703e-05

U = SimAnnealWalkSquare(m = mstar, T0 = T0, rate = coolingrate, Nsim = Nsim,
    plotbest = TRUE)$U
U = U[seq_along(U)%%500 == 0]
t = 500 * (1:floor(Nsim/500))
ggplot(data.frame(U, t), aes(x = t, y = U)) + geom_line(alpha = 0.7) + scale_x_continuous(labels = comma) +
    ggtitle(paste("Samples from U @ T0 =", T0, ", rate =", signif(coolingrate,
        1))) + theme_minimal() + theme(plot.title = element_text(size = 12))
```





Appendix: R functions for Problem #4

```r
pathdistance = function(path, distances) {
    stopifnot(is.matrix(distances) & min(distances) == 0)
    # first find the lenght of the first path
    m = length(path) - 1
    pd = distances[path[1], path[2]]  #distances is a matrix.
    if (m + 1 > 2)
        for (i in 2:m) {
            # assuming we have more then 2 cities we then add the distances for
            # additional paths
            pd = pd + distances[path[i], path[i + 1]]
        }
    pd  #return pd
}
switch_nonconsecutive = function(path) {
    # a potential proposal distribution exchange 2 non-consecutive nodes on the
    # path.
    n = length(path)
    stopifnot(n >= 4)
    # choose 2 cities randomly, then switch them
    inidices = sample(2:(n - 1), 2, replace = FALSE)
    # the next 3 lines just switch the 2 cities.
    temp = path[inidices[1]]
    path[inidices[1]] = path[inidices[2]]
    path[inidices[2]] = temp
    return(path)
}
vispath = function(path, cities, title = "Traveling Salesman Path") {
    # function to visualize a path.  This one is a lot more pretty and detailed
    # than what's required in your homework.  If you prefer to use the base
    # graphics, no problem.
    library(ggplot2)
    locations = as.data.frame(cities[path, ])  #put the x,y coordinates in the order of the path
    ggplot(locations, aes(x = x, y = y)) + geom_path() + geom_point(col = "red",
        size = 3) + theme_minimal() + ggtitle(title) + annotate("point", x = locations[1,
        1], y = locations[1, 2], color = "blue", size = 7)

    # How would you do this in base graphics? Try the following 2 lines:
    # plot(locations, main=title, type='o'); points(locations[1,], col='red',
    # pch=16);
}
placeCitiesCircle = function(m) {
    # create a vector of potential values for theta.  without [1:(m+1)] at the
    # end, the last value will be the same as the first... so we're only want
    # the first m+1 values.
    radius = 1
    thetas = seq(0, 2 * pi, by = 2 * pi/(m + 1))[1:(m + 1)]
    # translate to x,y values, put into a matrix
    cities = cbind(x = radius * cos(thetas), y = radius * sin(thetas))
    cities  #return this
}
MetropolisWalkCircle = function(m, T, Nsim = 10000, radius = 1, plotbest = TRUE) {
    # get locations of m+1 cities for our given m
    cities = placeCitiesCircle(m)
    distances = as.matrix(dist(cities))
    # set the initial path for U[1] path starts at 1, then goes to the rest
    # randomly
    path = c(1, sample(2:(m + 1), replace = FALSE), 1)
```

```
    # creat a vector for U, we'll write over all but U[1]
    U = rep(pathdistance(path, distances), Nsim)
    # remember which path is best.
    bestpath = path
    for (t in 2:Nsim) {
        newpath = switch_nonconsecutive(path)
        Uold = U[t - 1]
        Unew = pathdistance(newpath, distances)
        prob = ifelse(Unew < Uold, 1, exp(-(Unew - Uold)/T))
        if (runif(1) < prob) {
            path = newpath
            U[t] = Unew
        } else U[t] = Uold

        if (U[t] <= min(U))
            bestpath = path
    }
    if (plotbest)
        plot(vispath(bestpath, cities, paste0("T = ", T, ", Length = ", round(min(U),
            2))))
    # if the function doesnt see return() somewhere first, it will return
    # whatever's on the last line.
    list(U = U, bestpath = bestpath)
}
SimAnnealWalkCircle = function(m, T0, rate, Nsim = 10000, radius = 1, plotbest = TRUE) {
    # get locations of m+1 cities for our given m
    cities = placeCitiesCircle(m)
    distances = as.matrix(dist(cities))
    # set the initial path for U[1] path starts at 1, then goes to the rest
    # randomly
    path = c(1, sample(2:(m + 1), replace = FALSE), 1)
    # create a vector for U, we'll write over all but U[1]
    U = rep(pathdistance(path, distances), Nsim)
    # remember which path is best.
    bestpath = path
    T = T0
    for (t in 2:Nsim) {
        newpath = switch_nonconsecutive(path)
        Uold = U[t - 1]
        Unew = pathdistance(newpath, distances)
        prob = ifelse(Unew < Uold, 1, exp(-(Unew - Uold)/T))
        if (runif(1) < prob) {
            path = newpath
            U[t] = Unew
        } else U[t] = Uold

        if (U[t] <= min(U))
            bestpath = path

        T = T * (1 - rate)
    }
    if (plotbest)
        plot(vispath(bestpath, cities, paste0("T0 = ", T0, ", Rate = ", signif(rate,
            1), ", Length = ", round(min(U), 2))))
    # if the function doesnt see return() somewhere first, it will return
    # whatever's on the last line.
    list(U = U, bestpath = bestpath)
```

```
}
placeCitiesSquare = function(m) {
    # first value is at 0,0, others are in a unit box
    x = c(0, runif(m))
    y = c(0, runif(m))
    cbind(x, y)  #this last line gets returned
}
SimAnnealWalkSquare = function(m, T0, rate, Nsim = 10000, radius = 1, plotbest = TRUE) {
    # get locations of m+1 cities for our given m
    cities = placeCitiesSquare(m)
    distances = as.matrix(dist(cities))

    # set the initial path for U[1] path starts at 1, then goes to the rest
    # randomly
    path = c(1, sample(2:(m + 1), replace = FALSE), 1)
    # creat a vector for U, we'll write over all but U[1]
    U = rep(pathdistance(path, distances), Nsim)
    # remember which path is best.
    bestpath = path
    T = T0
    for (t in 2:Nsim) {
        newpath = switch_nonconsecutive(path)
        Uold = U[t - 1]
        Unew = pathdistance(newpath, distances)
        prob = ifelse(Unew < Uold, 1, exp(-(Unew - Uold)/T))
        if (runif(1) < prob) {
            path = newpath
            U[t] = Unew
        } else U[t] = Uold

        if (U[t] <= min(U))
            bestpath = path

        T = T * (1 - rate)
    }
    if (plotbest)
        plot(vispath(bestpath, cities, paste0("T0 = ", T0, ", Rate = ", signif(rate,
            1), ", Length = ", round(min(U), 2))))
    # if the function doesnt see return() somewhere first, it will return
    # whatever's on the last line.
    list(U = U, bestpath = bestpath)
}
```