Jeffrey Ding
UID: 104928991
CEE M20
May 12th, 2018

Homework 6

1. The Split-and-Average Problem

1.1 Introduction

The goal of this problem is to determine how many people are required in a group before it is more likely than not that any two of their birthdays occur in the same week.

1.2 Model and Methods

The script begins by creating a trials array that tracks the number of people in a group and a trialIndex variable determining the current trial:

```
trials = zeros(1, 10^4);
trialIndex = 1;
```

The script then runs a while loop that runs while there are still trials:

```
while (trialIndex <= 10^4)
...
end
```

Within the while loop, the script creates another array called group that will holding people in the group. It is of size 30 because it is highly unlikely there will be more than 53 people that don't have birthdays within the same week because 365/7 is roughly 52 and the worst case scenario is that 52 people have evenly spaced out birthdays and the last person has to be within the same week of someone else. The script also tracks groupIndex and a boolean called found that will determine when two people have birthdays on the same week:

```
group = zeros(1, 53);
groupIndex = 0;
found = 0;
```

Then, the script runs another while loop that runs when found is not true:

```
while(found ~= 1)
...
```

```
end
```

Within the while loop, the script increments groupIndex and adds a random birthday into the group, which is a number between 1 and 365:

```
groupIndex = groupIndex + 1;
group(groupIndex) = floor(rand(1, 1)*365) + 1;
```

Then, the script runs a for loop that iterates through all the previous elements of the group:

```
for k = 1:groupIndex-1
...
end
```

Running through the previous elements, the script first checks if the absolute difference between the current birthday and current previous day is less than 7. If that is false, there are two other situations that have to be checked. If the current day plus 7 is greater than 365, that means the script has to check for wrap around on calenders. So, the script checks for the same thing as before but first subtracts 365 from the current day. The script also checks if the current day - 7 is less than 1. The script checks for the same thing, but first adds 365 to the current day:

```
if abs(group(groupIndex)-group(k)) < 7
        trials(trialIndex) = groupIndex;
        trialIndex = trialIndex + 1;
        found = 1;
        break;
elseif group(groupIndex) + 7 > 365
        if abs(group(groupIndex)-365-group(k)) < 7
                trials(trialIndex) = groupIndex;
                trialIndex = trialIndex + 1;
                found = 1;
                break;
        end
elseif group(groupIndex) - 7 < 1
        if abs(group(groupIndex)+365-group(k)) < 7
                trials(trialIndex) = groupIndex;
                trialIndex = trialIndex + 1;
                found = 1;
                break;
        end
end
```

After running the while loop and filling the trials array, the script calculates the median using the median function:

median = median(trials);

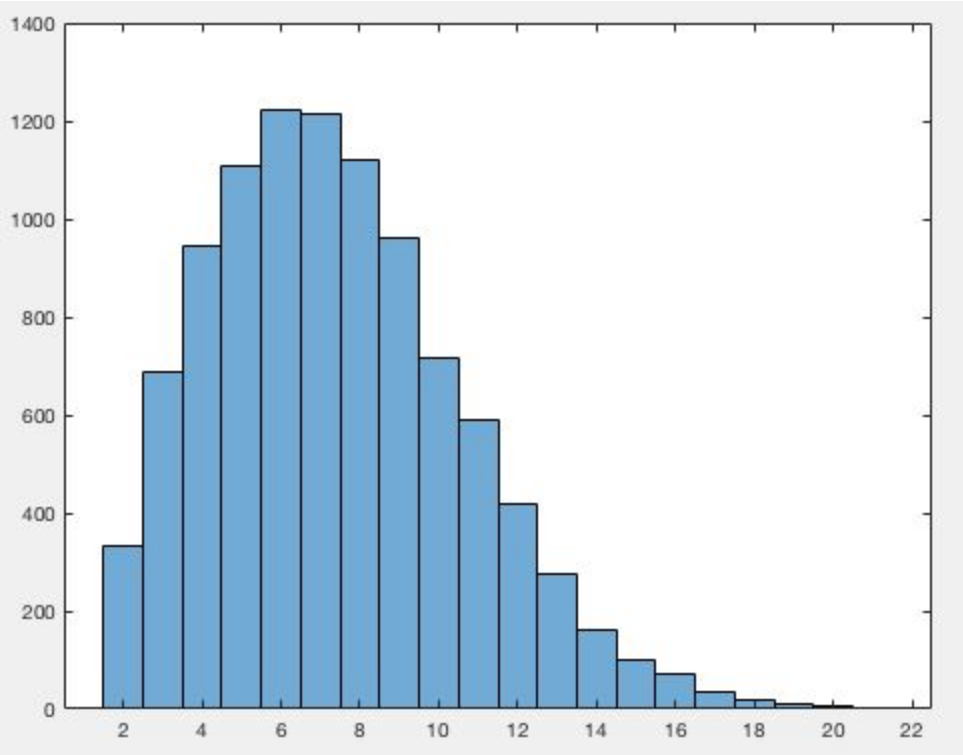
Then, the script prints the results:
fprintf("Median Number of People = %d\n", median);
histogram(trials);

1.3 Calculations and Results

When the program is executed the following output is printed to the screen:

Median Number of People = 7

The following graphs are created:



The output shows that the median is indeed 7.

1.4 Discussion

The histogram shows a skewed to the right graph, which means the median is the best way to interpret the graph. The median is obviously either 6 or 7 and it is indeed is 7 according to my calculations.

Considering the median of 7, if I were in a group larger than this, there is a greater than 50% chance of two people having a birthday in the same week. This number seems extremely reasonable to me because the calendar is essentially split into 14 day blocks that basically indicate that if two people's birthdays land within this area, it means they have birthdays within the same week.

As a matter of fact, I would expect this value to decrease if we accounted for that fact that not all birth dates are equally likely because more birthdays would be clustered to the days that are more common for births.

2. Simulated Annealing

2.1 Introduction

The goal of this problem is to find the shortest, fully-closed path between N randomly-positioned points on a 2D grid through a simulated annealing algorithm.

2.2 Model and Methods

The script begins by setting the number of cities to 10 and creating three arrays, an array x that holds random x coordinates, an array y that holds random y coordinates, and an array path that holds a random permutation of numbers from 1 to N (in the base case, 10), which represents the path:

```
N = 10;
x = rand(1, N);
y = rand(1, N);
path = randperm(N);
```

Then, the script determines the length of the initial path using the getPathDistance function:

```
distance = getPathDistance(x, y, path);
```

The getPathDistance function takes in three inputs, a vector, x, representing x coordinates, a vector y, representing y coordinates, and a vector path, a permutation of the values describing the order of visitation. It returns a scalar, dist, that represents the sum of the Euclidean distances between successive nodes in the path.

First, the function determines the number of nodes in the path and sets dist to 0:

```
nodes = length(path);
dist = 0;
```

Then, the script iterates through the first N-1 visitations, set a node a as the current node and a node b as the previous node. Then it calculates the distance and adds that to the current dist:

```
for k = 2:nodes
   a = path(k-1);
   b = path(k);
   dist = dist + sqrt((x(b)-x(a))^2 + (y(b)-y(a))^2);
end
```

After the for loop, the function has to calculate the wrap the return trip from the last node to the first node like this:

```
dist = dist + sqrt((x(nodes)-x(1))^2 + (y(nodes)-y(1))^2);
```

Back to the script! The script creates a array called shortest that tracks the shortest distances as a function of iteration:

```
shortest = zeros(1, 1379);
shortestIndex = 0;
```

Then, the script begins simulated annealing:

```
T = 1000;
while (T > 1)
...
end
```

First, the shortestIndex variable is incremented and the current shortest distance is added to the shortest array:

```
shortestIndex = shortestIndex+1;
shortest(shortestIndex) = distance;
```

Then, the script creates a pathNew and swaps random nodes as an exploration of a different path:

```
pathNew = path;
```

```
firstIndex = floor(rand(1, 1)*N) + 1;
secondIndex = firstIndex;
while firstIndex == secondIndex
        secondIndex = floor(rand(1, 1)*N) + 1;
end
tempNode = pathNew(firstIndex);
pathNew(firstIndex) = pathNew(secondIndex);
pathNew(secondIndex) = tempNode;
```

Then, the function calculates the new distance using the getPathDistance function and calculates the difference between the new path and the current path distances:

```
distNew = getPathDistance(x, y, pathNew);
deltaL = distNew - distance;
```

Then, the script evaluates whether or not to accept the new path. First, in order to encourage random guessing when T is large, the script creates a variable, p, an exploration metric that sometimes allows the program is explore random paths:

```
c = 1000;
p = exp(-c*deltaL/T);

if deltaL < 0 || p > rand(1, 1)
        path = pathNew;
        distance = distNew;
end
```

Then, at the end of the while loop, cool the simulation:

```
T = T*(1-0.005);
```

After the while loop, the script reassigns the x and y values into a new order of x and y coordinates based on the shortest path calculate:

```
newX = zeros(1, N+1);
newY = zeros(1, N+1);
for k = 1:N
   currentIndex = path(k);
   newX(k) = x(currentIndex);
   newY(k) = y(currentIndex);
end
newX(N+1) = x(path(1));
newY(N+1) = y(path(1));
```

Then, the script plots the results, which is a plot of a shortest path and a graph indicating the shortest distance at each iteration:

```
figure(1);
plot(newX, newY, 'o-');
figure(2);
plot(shortest);
```

2.3 Calculations and Results

When the program is executed the following graphs are created:

Figure 1

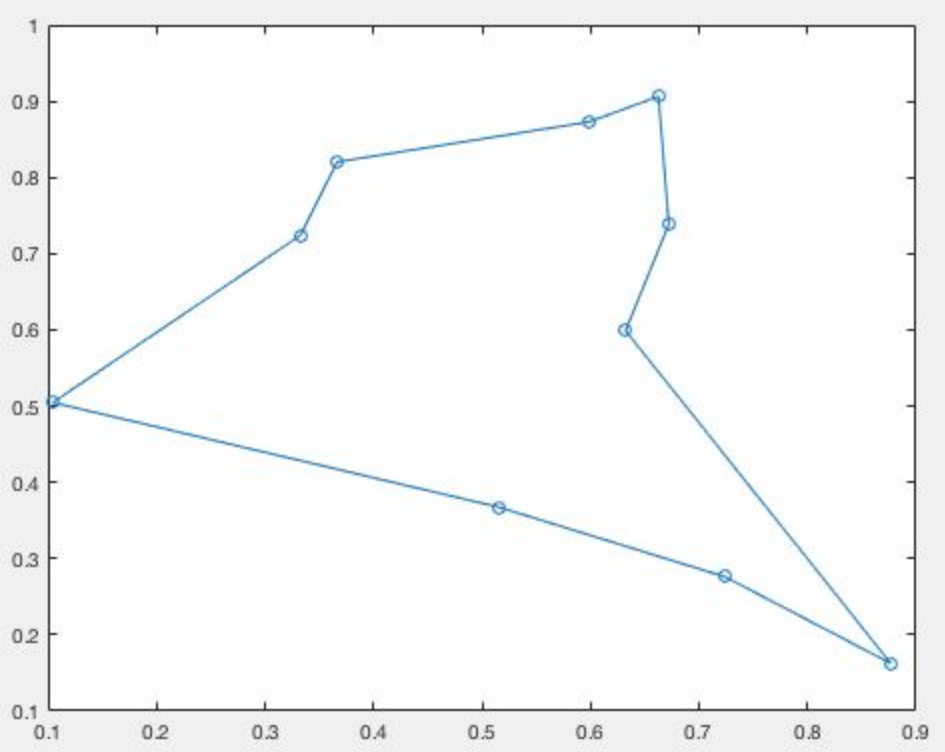
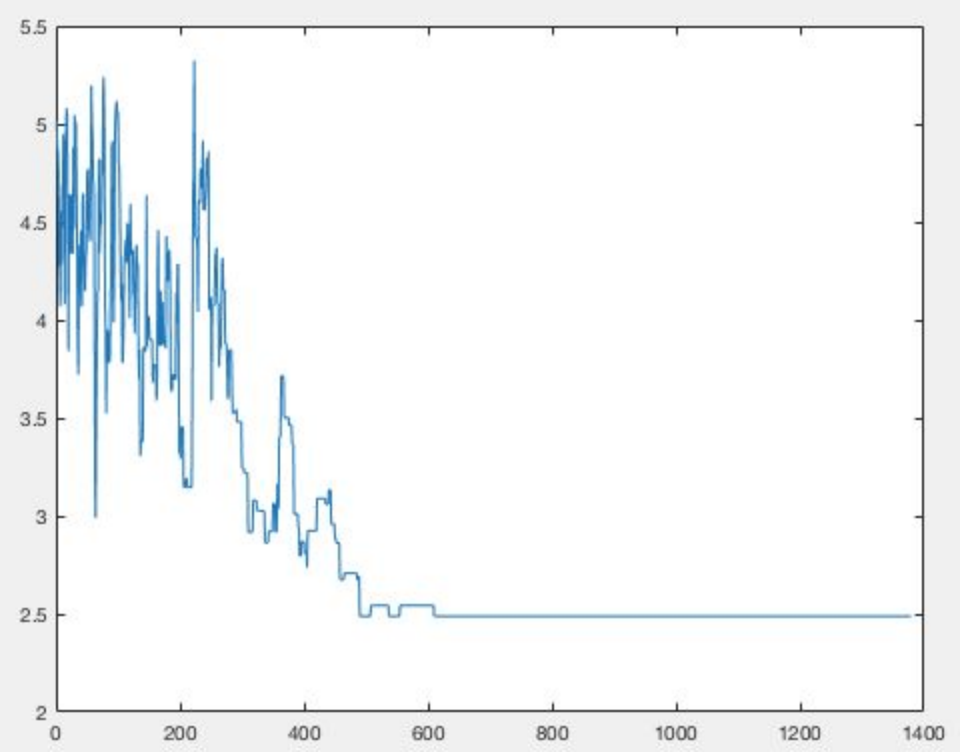


Figure 2



The outputs look like the shortest path was found!

2.4 Discussion

The outputs look like the shortest path was found in the current simulation. However, this is based on an educated guess because the final path has the shortest distance recorded but there is a possibility that it is not the shortest path.

My solution does not always find the shortest path because (1) there is always a very very very small possibility that a new path is created due to the exploration metric that makes the final value different from the shortest path or (2) that the exploration metric took the path is a relative minimum and that the simulation was unable to escape from that point.

As I increase the value of N, the solution produced by my method has a higher probability of generating error. This is due to the more and more possibility of paths that can be generated, thus infinitely more paths to explore.

Even though this approach is slightly flawed, I can not find a better combination of lambda and c that improves the behavior when N > 10. However, I believe this is unnecessary because the algorithm creates a relatively good estimate of the shortest path.