Candidate's Campaign Strategy

And

Ambulance Routes in Manhattan

Control Number: #841

Table of Contents:

vi. Sensitivity Analysis

vii. Strength and Weakness

viii. Citations

5. Appendix

1. Summary for Problem I

In this problem, we determine a strategy to divide a candidate's campaign finance resources in order to maximize his/her probability of winning the presidential election. In devising this plan, we consider the results from current polls indicating how well the candidate is doing compared to the opponent. The poll results include each person's certainty factors towards the candidates.

To approach this problem we follow this method: $money \rightarrow person \rightarrow state \rightarrow nation$. First, we model the relationship between the amount of money per citizen one candidate outspends the other candidate and the citizen's certainty factor towards the candidate who invests more money. The model is based on the $\arctan$ curve. We use data from the 2012 presidential election to determine the parameters of this curve. We then randomly generate data sets of citizens' certainty factors in order to simulate models for each state. For all states, we calculate how much money one candidate has to outspend the other candidate in order to win the state election. We rank the states by efficiency of investing money to winning that state. In order to make an optimized financial plan for winning the election, we conclude that the candidate needs to maximize the Electoral College votes relative to money spent. We continue to add states into our simulation until reaching 270 votes in total, which are enough to win the national election.

After this process, we are able to give the candidate a list containing the name of states and the least amount of money the candidate should outspend the opponent for each state.

To test our model, we use data from the ongoing 2016 United States presidential election. Based on the level of support for Hillary Clinton in polls for the 51 states, we divide her financial resources such that she outspends the least amount of money to win the election.

2. Summary for Problem II

In problem 2, we design efficient ambulance routes for Manhattan so that hospitals can send their ambulances to car crash sites in the shortest time.

To achieve this, we first build a vertex figure modeling Manhattan, including major crossings being the vertices and roads being the edges. We also add hospitals that can operate emergency surgeries into our vertex figure shown as vertices.

We assign values to the edges, such that the values represent the time an ambulance needs to pass the road completely. By using Dijkstra's Algorithm, we find the closest hospital to each vertex and include the vertices within the area of which the closest hospital takes charge. In this way, we partition Manhattan into 8 areas in terms of the results (there are 8 emergency hospitals in Manhattan), and the hospital is responsible for reacting to car accidents happening in its area. We also considered the situations when the closest hospital runs out of ambulances. Finally, we made a static to show the car crash and hospital response in Manhattan.

3. Model for Problem I

3-i Introduction

The United States goes through a presidential election every four years, during which two candidates run for office in the national election. The 51 states in the United States have different number of votes: the number of votes a state has is determined by the state's population. People vote in state elections, and all the state's electoral votes go to the candidate who wins the state election. If a candidate wins over 270 out of the 538 electoral votes in the national election, this candidate becomes the president.

While there are many strategies candidates can use to gain support, their financial strategy is the most important one. The amount of money they invest in a state has a direct impact on their success in the state's election. In this problem, we are trying to find a strategy that gives the candidate an optimized plan to distribute his/her money to the states.

3-ii Assumptions and Justifications

1) We know current poll information. People who participated in the poll will eventually vote in the election.
2) If someone is swaying between candidate A and candidate B, we split the swaying factor into halves for both candidates. For instance, if a person has a certainty factor of 18% towards candidate A and a certainty factor of 22% towards candidate B, we assume that this person has a certainty factor of 48% towards candidate A and a certainty factor of 52 % towards candidate B. Thus, everyone has only two certainty factors, either tending to candidate A or candidate B.

3) All people's certainty factors for the candidates will be influenced by how much money one candidate outspends on them than the other one.
4) If a person has a 50% certainty factor towards candidate A and a 50% certainty factor towards candidate B, we just need to spend a little bit more money on this person than our opponent in order to win this person's vote. We assume this "a little more money" is infinitely small, which means that a 50% certainty factor would win this person's vote.
5) For the same reason, if 50% of people in a state have 50% certainty factors for our candidate, we assume that we win this state's votes.
6) According to the electoral process in the United States, if a candidate wins a state, all the state's electoral votes go to this candidate.

3-iii Symbols

m	Money for a person
M	Money for a state
c	A person's certainty factor towards candidate A (our candidate)
1-c	A person's certainty factor towards candidate B
k	Number of people's certainty factor data sets for the state
E	Efficiency of money
N	Number of electoral votes a state has
P	Population of a state

Table 3-1: symbols used in Problem I

3-iv Building the Model

Since we assume that all people's certainty factors will be influenced by the amount of money one candidate outspends the other one (assumption 2), we use a function to model this positive correlation.

Since a person's certainty factor will be harder and harder to increase and has an asymptote at $y = 100\%$, we model the relationship between money and certainty factor using a $\arctan q$ curve[1].

$$f(x) = \frac{2}{p}\arctan(Ax) \qquad (1)$$

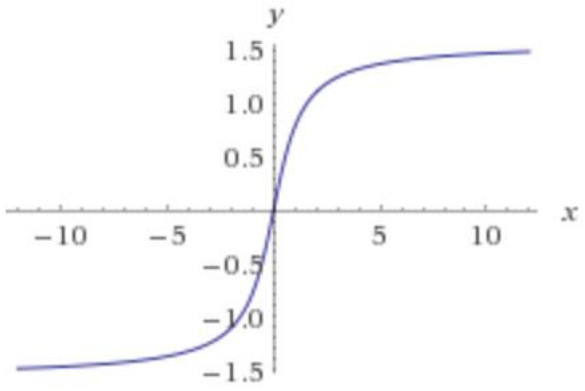

Figure 1. The curve of general $\arctan$ curve

We determine the constant A in (1) by using data from past presidential elections. Precisely, we use data from the 2012 presidential election between Barack Obama and Willard Romney.

[1] http://www.wolframalpha.com/input/?i=y%EF%BC%9Darctan(x)

For the state of Florida which has a population of 18,801,310[2], we find that Romney outspent 998,973 dollars than Obama[3], and Romney's supporting percentage in this state raised from 40% to 49%[4]. Since we divide the swaying factor in half, Romney's supporting rate in Florida actually raised from 48.5% to 49.5%.

We solve for A:

$$f^{-1}(49.5\%) - f^{-1}(48.5\%) = \frac{998973}{18801310}$$

$$A = 0.005161$$

Thus, we conclude that the relationship between money and a person's certainty factor towards one candidate can be model by[5]:

$$f(m) = \frac{2}{p}\arctan(0.005161' m) \qquad (2)$$

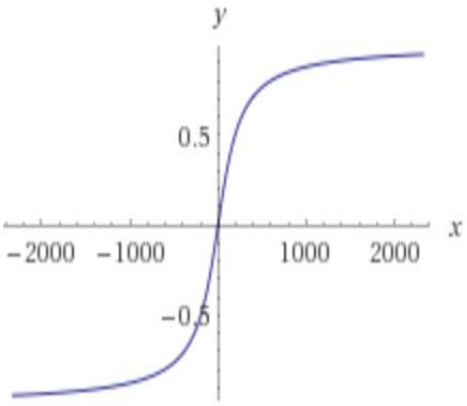

Figure 2. The curve of $f(m)$

[2] http://population2016.com/population-of-florida-in-2016.html
[3] http://www.politico.com/story/2012/11/early-voting-results-2012-083176
[4] http://elections.nbcnews.com/ns/politics/2012/all/president/#.WABloDJVaT8
[5] http://www.wolframalpha.com/input/?i=y%3D2*arctan(0.005161*x)%2Fpi

However, people do not directly decide who the president shall be. They vote in state elections, and it is the state that has electoral votes in the national election. The number of votes each state gets depends on its population. Thus, we generate people's certainty factor datasets for each state, according to the state's population. To simplify the problem, we generate $\left[\frac{P_{state}}{100000}\right]$ datasets for the state's population, P_{state}, and times 100,000 in the end while calculating the campaign money to spend on this state. The table of states' populations and the number of electoral votes they have is included in Appendix 1.

We randomly generate data of people's certainty factors--c--for our candidate.

In the assumption, we assume that if 50% of the people in a state have 50% certainty factors for our candidate, then we win the state. Thus, our goal here is to find the least amount of money we need to outspend our opponent on a state so that we can have 50% of the people in the state having 50% certainty factors for our candidate.

For example, if a state has 6 sets of data for people's certainty factors--$c_1, c_2, c_3, c_4, c_5, c_6$ in descending order--then our function would be like:

$$M = \min \sum_{i=1}^{3} \left(f^{-1}(50\%) - f^{-1}(c_i) \right) \qquad (3)$$

This will give us the amount of money we need to outspend the other candidate for this state to win this state. A positive M means that we need to outspend the other candidate in this state, while a negative M means that our opponent has to outspend us to a certain amount but still not able to win this state.

We do the same process for all 51 states. We also take into account the fact that the 51 states have different numbers of electoral votes. We want to minimize our spending while getting over 270 votes to win the election. When determining which states invest money in, we consider the efficiency of our money:

$$E_{state} = \frac{M_{state}}{N_{state}} \quad (4)$$

The smaller the E_{states} is, the less we need to spend on a state to get one electoral vote from the state. We spend campaign money on states with smaller E_{states} , till we are able to get over 270 votes.

Thus, we are able to have a plan of spending the least money while having our candidate win over 270 votes (code for realizing this model is in Appendix 2).

3-v Results from the Model

The result from our model is that, based on the data sets we generate, we need to outspend 96717 dollars than our opponent in order to win the election. This breaks down to each state:

Efficient State	State	Electoral votes	Money
1	Texas	38	-1865
2	New York	29	-831
3	California	55	-1502
4	Florida	29	1866

5	Illinois	20	4418
6	Pennsylvania	20	8276
7	Ohio	18	10766
8	Michigan	16	14030
9	Georgia	16	17828
10	North Carolina	15	20429
11	New Jersey	14	23303

Table 2-2: Results table for which states to invest money in and how much to outspend

Among all the states, these 11 states are the ones with the smallest E_{state} s while giving us over 270 electoral votes to win the election. The full result table is included in Appendix 3.

3-vi Sensitivity Analysis

People's certainty factors are the only variables that affect our result.

3-vii Testing the Model

To test our model, we use data from the ongoing 2016 U.S. presidential election between Donald Trump and Hillary Clinton. We record Hillary Clinton's supporting percentages in each state, and view them as the average people's certainty factor towards her in these states (table of Hillary's supporting percentages in each state is included in Appendix 4)[6]. By putting these

[6] http://www.upi.com/Top_News/US/2016/10/03/UPICVoter-state-polls-Donald-Trump-gains-support-in-all-but-5-states/6881475512049/

certainty factors into our model, we get the optimized plan for distributing Hillary's finance resources.

Our result is that Hillary Clinton needs to outspend -89,523,541 dollars than Donald Trump to win the election. This breaks down into states:

Efficiency	State	Outspend trump
1	Washington D.C.	-36928082
2	Vermont	-8193297
3	Hawaii	-10735863
4	Rhode Island	-4577022
5	Delaware	-2741457
6	Maine	-2394096
7	Maryland	-5281931
8	Massachusetts	-3970813
9	Connecticut	-2325359
10	New Jersey	-3601180
11	New York	-4809224
12	Illinois	-2256864
13	Washington	-1257350
14	California	-4654121
15	Oregon	-122128
16	Michigan	121363
17	New Mexico	60776
18	Wisconsin	241970
19	Minnesota	361831

20	Florida	2130814
21	Ohio	1408491

Table 2-3: Results table for which states to spend money on and how much to spend for Hillary

These states will give Hillary 275 votes, which is enough to win the election. The result table of all the states is included in Appendix 5.

3-viii Strength and Weakness

In this problem, we are devising a strategy that gives the candidate an optimized plan to distribute his/her money to the states. Using our model, we are able to determine which states our candidate should invest in, and how much he/she should outspend the other candidate in these states.

Strength:

1) Our model is not just based on what our candidate should do: it's based on how much more our candidate should do than the other candidate. Comparing to models that give fixed amounts of campaign money to the states, our model is more suitable to the competitive environment of the election.

Weaknesses:

1) Our $f(m)$ model is not based on a lot of research. Although we use data to test the parameters of the equation, a more advanced model is needed to model the complex relationship between money and people's certainty factors.

3-ix Citations

[1] Jay Belanger, *Write Right for the American Mathematical Contest in Modeling*, Higher Education Press, 2013.

[2] John Goerzen, *Foundations of Python Network Programming*, Publishing House of Electronics Industry, 2007.

4. Model for Problem II

4-i Introduction

When a major car collision occurs, it is the hospitals' responsibility to dispatch ambulances to reach the site as soon as possible. However, in cities with heavy traffic and high frequency of collision cases, like Manhattan, it is necessary to design optimized ambulance routes so that the hospitals can accomplish the task of reaching crashing sites efficiently. In this problem, we are asked to develop such a plan for Manhattan.

4-ii Assumptions and Justifications

1) Ambulances come out of hospitals in Manhattan that have abilities to operate emergency surgeries, and go back to the same hospitals.

2) Because ambulance dispatch is not necessary if no people are hurt in the car accident, we do not consider the collision where no harm occurred.

3) To simplify the problem, we ignore the fact that roads in Manhattan are one-way roads. We assume that ambulances can go both ways.

4) We assume that every data point provided by the problem has an equal chance of having a car collision.

5) To simplify the problem, we assume that the ambulance only needs to get to the crossing near the actual collision site.

4-iii Symbols

d	Road length
t	Time to cover the road, in seconds
P_i	The probability of a car crash occurring around a point
N_i	The number of actual car collision happened around that point
T	Duration of time in the record, in minutes

Table 4-1: Symbols used in Problem II

4-iv Building the Model

Firstly, we utilize the car accident data provided by the problem. Since we only send out ambulances when someone is injured or dead, we delete accidents that do not include any injuries or deaths.

We generate a vertex figure based on the map of Manhattan, with major crossings being the vertices and the roads being the edges. We also search for hospitals in Manhattan that can perform emergency surgeries, and find 8 hospitals in total (hospitals' names and locations are included in Appendix 6). We add them onto the vertex figure as vertices.

Figure 3. Vertex Figure based on the Map of Manhattan

We also assign values to the edges of the graph such that the values reflect the time an ambulance needs to pass this road completely. Based on the research, we find out that the ambulance speed limit in Manhattan is 25 mph[7]. Thus we assign values of t to the edges (value table is in Appendix 7):

$$t_{road} = \frac{d_{road}}{25\ mph} \tag{7}$$

[7] http://www.nytimes.com/2014/11/08/nyregion/a-lower-speed-limit-takes-effect-in-the-city-lower-speed-maybe-not-much.html

For each vertex representing the crossing, we determine the closest hospital (spending the least time to get to) to that vertex using Dijkstra's Algorithm. Then we include the crossings into the range of the closest hospital found by the Algorithm. Since 8 hospitals are under our consideration, there are 8 areas partitioning Manhattan, which correspond to the 8 hospitals mentioned in Appendix 6.

Figure 4. Partitioned map of Manhattan

According to this map, the hospitals should respond to and only to car collisions inside their assigned range.

In addition, Dijkstra's Algorithm gives us the shortest routes that the ambulances should take from the hospital to the crossing. For example, if a car crash happens at vertex 210, our model will tell us to drive in the following route: 318 => 155 => 154 => 172 => 187 => 202 => 210, which takes 119 seconds on the road to reach the crashing site and back.

In the assumption, we assume that every data point provided by the problem has an equal chance of having a car crash. We assign values to each vertex on the vertex figure, such that the value represents the probability of a car collision occurring near this vertex. We calculate this probability by dividing the actual number of car collisions happened in a zip code area over the duration of time T, which, in terms of the data provided by the problem, is:

$$T = 1550 days \acute{} 24^{hours}/_{day} \acute{} 60^{\min}/_{hour} = 2232000 \min \quad (5)$$

Then, we are able to get the possibility, P_i, of a car crash happening near a vertex:

$$P_i = \frac{N_i}{T} \quad (6)$$

To simplify the problem, we separate Manhattan into $19 \acute{} 19$ areas of equal sizes, in terms of its longitude and latitude. The data points i all fall on the areas, and we calculate the amount of car crash N_i in each area. We assign vertices values of P_i according to the area the vertices belong to. The results are shown in Appendix 10. Thus, with the output of a probability

distribution, we build a static model, which simulates the car crashes and hospital responses in Manhattan (Appendix 11).

Here are our results:

246th minutes

280 car accidents

321 ambulances sent by all the hospitals

286th minutes

262 car accidents

321 ambulances sent by all the hospitals

909th minutes

299 car accidents

321 ambulances sent by all the hospitals

We also consider the case when the hospital in the petitioned area where the ambulances are running out. According to Poisson Distribution,

$$P(k) = \frac{l^k e^{-l}}{k!} \tag{7}$$

the expected value of n car crashes happening at the same time can be calculated as the following:

$$E\left(\prod_{a=0}^{n}(X - a)\right) = l^n \tag{8}$$

If the number of car crashes happening in an area exceeds the number of ambulances provided by a certain hospital, we will compare the time that the ambulance spends from the

crashing site to the other crashing site (including dropping the patient to the hospital during the process) with the one that the ambulance spends from another hospital in Manhattan to the site.

4-v Results from the Model

In this problem, we are asked to design sufficient ambulance routes for Manhattan hospitals. Our result is the partitioned map of Manhattan and the static model. When a car collision happens, the hospital in the range area should be responsible for sending an ambulance to the site. Also, for most of the vertices, it is faster for the hospital in the area to dispatch an ambulance when available.

4-vi Sensitivity Analysis

For our model, the result can be affected by the following factors:

1) Traffic jam in Manhattan. Traffic jam is uncontrollable and is always changing, and it will affect the actual time an ambulance needs to reach the crashing site as well as the shortest route the ambulance can take.
2) The probability of a car crash happening. If the probability is timed by a constant c, which gives us cl as the probability, then the expected value of n car crashes occurring at the same time can be calculated as $E\left(\prod_{a=0}^{n}\left(X - a\right)\right) = \left(c\ l\right)^n$. We can see that the probability is multiplied by c^n.

4-vii Strength and Weakness

Strengths:

1) Our model allows every point in Manhattan to be covered—wherever a collision happened, we have a plan.

2) We take into consideration that in a crowded city like Manhattan, traffic jam could be a big factor in influencing the traveling time on the roads.

Weaknesses:

1) We do not consider every road in Manhattan in our model, and we do not consider the fact that most roads in Manhattan are one-way roads. Both can influence the shortest route an ambulance can take.

2) When we consider the concentration of car crashes, we simplify the problem by separating Manhattan into $19' 19$ areas of equal size, which can cause some inaccuracy.

4-viii Citations

[1] NYC Hospitals Map and List: https://www.newyorkled.com/nyc-hospitals-map-and-list.htm#12/39.8418/-73.9032

[2] Poisson distribution, Wikipedia, https://en.wikipedia.org/wiki/Poisson_distribution

5. Appendix

Appendix 1: table of states' populations and the amount of electoral votes they have

Rank	State	Population	Electoral votes
	All United States	308745538	
1	California	37253956	55
2	Texas	25145561	38
3	New York	19378102	29
4	Florida	18801310	29
5	Illinois	12830632	20
6	Pennsylvania	12702379	20
7	Ohio	11536504	18
8	Michigan	9883640	16
9	Georgia	9687653	16

10	North Carolina	9535483	15
11	New Jersey	8791894	14
12	Virginia	8001024	13
13	Washington	6724540	12
14	Massachusetts	6547629	11
15	Indiana	6483802	11
16	Arizona	6392017	11
17	Tennessee	6346105	11
18	Missouri	5988927	10
19	Maryland	5773552	10
20	Wisconsin	5686986	10
21	Minnesota	5303925	10
22	Colorado	5029196	9
23	Alabama	4779736	9
24	South Carolina	4625364	9
25	Louisiana	4533372	8
26	Kentucky	4339367	8
27	Oregon	3831074	7
28	Oklahoma	3751351	7
29	Connecticut	3574097	7
30	Iowa	3046355	6
31	Mississippi	2967297	6
32	Arkansas	2915918	6
33	Kansas	2853118	6
34	Utah	2763885	6
35	Nevada	2700551	6
36	New Mexico	2059179	6
37	West Virginia	1852994	5
38	Nebraska	1826341	5
39	Idaho	1567582	4
40	Hawaii	1360301	4
41	Maine	1328361	4
42	New Hampshire	1316470	4
43	Rhode Island	1052567	4
44	Montana	989415	3
45	Delaware	897934	3
46	South Dakota	814180	3
47	Alaska	710231	3
48	North Dakota	672591	3
49	Vermont	625741	3
50	Washington, D. C.	601723	3
51	Wyoming	563626	3

Appendix 2: Python code for Problem I

```
#Oct12 AoCMM Election
import random
import math
global budget
global money
global count
money=0
budget=0
person=1/2
state=[]
count=0
states=0
statet=0
votes=0
def ceil(number):#Gauss+1
	return int(number)+1
def moneytovote(money):
	return 2/math.pi*math.atan(money/0.005161)#function
def votetomoney(vote):
	return math.tan(vote*math.pi/2)*0.005161#inverse function
halfmoney=votetomoney(1/2)
def personmoney():#Apply to one person
	person=random.randrange(40,60)/100
	state.append(person)
	return halfmoney-votetomoney(person)
statesmoney=[]
def statemoney(numberofperson):#Apply to a state & Calculate the least amount of money needed
	count=1
	state=[]
	global state
	while count<numberofperson:
		statesmoney.append(personmoney())
		count=count+1
	sorted(statesmoney)
	global budget
	for a in range(1,ceil(len(statesmoney)/2)):
		budget=budget+statesmoney[a]
	return budget
countryinfo=[37253956,25145561,19378102,18801310,12830632,12702379,11536504,9883640,968765
3,9535483,8791894,8001024,6724540,6547629,6483802,6392017,6346105,5988927,5773552,5686986,5
303925,5029196,4779736,4625364,4533372,4339367,3831074,3751351,3574097,3046355,2967297,291
5918,2853118,2763885,2700551,2059179,1852994,1826341,1567582,1360301,1328361,1316470,10525
67,989415,897934,814180,710231,672591,625741,601723,563626]#number of voters in a state
countrymoney=[]#moneyneeded to win a state
countryorder=[]#effeciency
country=[]#sorted effeciency
statesname=["CA", "TA", "NY", "FL", "IL", "PA", "OH", "MI", "GE", "NC", "NJ", "VA", "WA", "MA",
"IN", "AZ", "TE","MO", "MD", "WI", "MN", "CL", "AL", "SC", "LO", "KE", "OR", "OK", "CO", "IO",
```

```
"MS", "AR", "KA", "UT", "NV", "NM", "WV", "NE", "ID", "HA", "ME", "NH", "RI", "MO", "DE",
"SD", "AK", "ND", "VE", "WD", "WY"]
stateschosen=[]
countryvotes=[55,38,29,29,20,20,18,16,16,15,14,13,12,11,11,11,11,10,10,10,10,9,9,9,8,8,7,7,7,6,6,6,6,6,
6,6,5,5,4,4,4,4,4,3,3,3,3,3,3,3,3]
while statet<=50:
        countrymoney.append(statemoney(int(countryinfo[statet])/100000))
        #print(statesname[statet])
        #print(state)
        statet=statet+1
        #print(statet/100)
while states<=50:
        countryorder.append(countrymoney[states]/countryvotes[states])
        states=states+1
        #print((states+50)/100)
country=sorted(countryorder)
strategy=[]
def strategywin():#generate money needed to win the vote
        states=0
        global count
        while count<270:
                strategy.append(countrymoney[countryorder.index(country[states])])#1. find smallest
effeciency 2. find its place in order 3.append its money
                stateschosen.append(statesname[countryorder.index(country[states])])
                count=count+countryvotes[countryorder.index(country[states])]
                states=states+1
        global money
        money=sum(strategy)


strategywin()
#print(state)
print("You need",money*100000,"more money than your opponent to win the election.")
for x in range(1,len(strategy)+1):
        print("state:",stateschosen[x-1])
        print("money:",strategy[x-1]*100000)
        print("votes:",countryvotes[countrymoney.index(strategy[x-1])])
        votes=votes+countryvotes[countrymoney.index(strategy[x-1])]

print("You get",count,"votes.")
print(votes)
print("overall result: ")
for x in range(0,51):
        print("The",x+1,"effecient state:")
        print("state:", statesname[countryorder.index(country[x])])
        print("money:", countrymoney[countryorder.index(country[x])]*100000)
        print("votes:", countryvotes[countryorder.index(country[x])])
#for x in range(0,50):
        #print("state:",statesname[x])
        #print("money:",countrymoney[x]*100000)
        #print("votes:",countryvotes[countrymoney.index(strategy[x])])
```

Appendix 3: Full result table for Problem I

Efficient State	State	Electoral votes	Outspend Money
1	Texas	38	-1865
2	New York	29	-831
3	California	55	-1502
4	Florida	29	1866
5	Illinois	20	4418
6	Pennsylvania	20	8276
7	Ohio	18	10766
8	Michigan	16	14030
9	Georgia	16	17828
10	North Carolina	15	20429
11	New Jersey	14	23303
12	Virginia	13	26986
13	Washington	12	30996
14	Massachusetts	11	35055
15	Indiana	11	39685
16	Arizona	11	44714
17	Tennessee	11	50126
18	Missouri	10	56035
19	Maryland	10	61397
20	Wisconsin	10	66261
21	Minnesota	10	70433
22	Colorado	9	75115
23	Alabama	9	80058
24	South Carolina	9	84872
25	Louisiana	8	89287
26	Kentucky	8	93648
27	Oregon	7	97843
28	Oklahoma	7	101809
29	Connecticut	7	105717
30	Iowa	6	109413
31	Mississippi	6	112620
32	Arkansas	6	115784
33	Kansas	6	119204
34	Utah	6	122487
35	Nevada	6	126068
36	New Mexico	6	129300
37	West Virginia	5	132270
38	Nebraska	5	135337
39	Idaho	4	137895
40	Hawaii	4	140457
41	Maine	4	142840
42	New Hampshire	4	145439
43	Rhode Island	4	148095
44	Montana	3	151041
45	Delaware	3	153628
46	South Dakota	3	156358

47	Alaska	3	159223
48	North Dakota	3	161984
49	Vermont	3	164498
50	Washington, D. C.	3	167141
51	Wyoming	3	169903

Appendix 4: Hillary's supporting rates in each state Oct. 2016

State	Hillary supporting rate
California	0.568
Texas	0.392
New York	0.570
Florida	0.463
Illinois	0.535
Pennsylvania	0.464
Ohio	0.476
Michigan	0.498
Georgia	0.425
North Carolina	0.455
New Jersey	0.554
Virginia	0.453
Washington	0.520
Massachusetts	0.559
Indiana	0.406
Arizona	0.420
Tennessee	0.381
Missouri	0.432
Maryland	0.576
Wisconsin	0.496
Minnesota	0.494
Colorado	0.483
Alabama	0.360
South Carolina	0.408
Louisiana	0.386
Kentucky	0.366
Oregon	0.502
Oklahoma	0.320
Connecticut	0.536
Iowa	0.475
Mississippi	0.397
Arkansas	0.358
Kansas	0.383
Utah	0.282
Nevada	0.471
New Mexico	0.499
West Virginia	0.344
Nebraska	0.369
Idaho	0.319
Hawaii	0.636
Maine	0.537
New Hampshire	0.484
Rhode Island	0.567
Montana	0.388
Delaware	0.542
South Dakota	0.400

Alaska	0.403
North Dakota	0.359
Vermont	0.610
Washington, D. C.	0.789
Wyoming	0.294

Appendix 5: Full result table of Hillary's model

Efficiency	State	Outspend trump
1	Washington D.C.	-36928082
2	Vermont	-8193297
3	Hawaii	-10735863
4	Rhode Island	-4577022
5	Delaware	-2741457
6	Maine	-2394096
7	Maryland	-5281931
8	Massachusetts	-3970813
9	Connecticut	-2325359
10	New Jersey	-3601180
11	New York	-4809224
12	Illinois	-2256864
13	Washington	-1257350
14	California	-4654121
15	Oregon	-122128
16	Michigan	121363
17	New Mexico	60776
18	Wisconsin	241970
19	Minnesota	361831
20	Florida	2130814
21	Ohio	1408491
22	Pennsylvania	2076192
23	Cleveland	1008139
24	Texas	5667205
25	North Carolina	2562376
26	Virginia	2668794
27	New Hampshire	950266
28	Iowa	1465017
29	Georgia	4101212
30	Nevada	1689489
31	Missouri	3752720
32	Arizona	4346454
33	Indiana	5017593
34	South Carolina	4923074
35	Tennessee	6163572
36	Louisiana	5939427
37	Alabama	7079649
38	Kentucky	6821938
39	Mississippi	5437567
40	Kansas	6074204
41	Arkansas	7164867
42	Oklahoma	8723994
43	Nebraska	6691902
44	West Virginia	7752164
45	Utah	10183933

46	Alaska	5158551
47	South Dakota	5298537
48	Montana	5849085
49	Idaho	8763595
50	North Dakota	7122301
51	Wyoming	9732412

Appendix 6: Names of hospitals and their locations

Name	Address
Lenox Hill Hospital EMS	100 East 77th Street, New York, NY, 10021
New York Presbyterian Hospital EMS	525 East 68th Street, New York, NY, 10021
St. Luke's / Roosevelt Hospital Center EMS	1000 Tenth Avenue, New York, NY, 10019
Bellevue Hospital Center	462 First Avenue, New York, NY,
Metropolitan Hospital Center	1901 1st Avenue, Manhattan
Mount Sinai Beth Israel	First Avenue at 16th Street 10003
N.Y.U. Langone Medical Center	550 First Avenue, Manhattan
NewYork–Presbyterian/Lower Manhattan Hospital	170 William Street, Manhattan

Appendix 7: Assigned values to the edges

1,2,10.0	32,33,15	60,76,20	88,104,30
1,126,525.0	32,39,10	61,62,10	89,90,10
2,3,13.0	33,34,15	61,77,20	89,105,30
3,4,12.0	33,40,10	62,63,10	90,91,10
3,15,30.0	34,41,10	62,78,20	90,106,30
4,5,15.0	35,47,10	63,64,20	91,92,10
4,16,30.0	35,48,14	63,79,20	91,107,30
5,6,7.0	36,37,12	64,80,25	92,93,10
5,17,30.0	36,67,60	65,66,10	92,108,30
6,7,14.0	37,50,7	65,81,42	92,315,8
6,12,15.0	38,39,15	66,67,15	93,94,10
7,8,12.0	38,54,30	66,82,40	93,109,30
7,19,30.0	39,40,15	66,83,45	94,95,13
8,9,15.0	39,55,30	67,68,15	95,96,15
8,13,18.0	40,41,15	67,84,40	95,110,22
9,10,14.0	40,56,30	68,69,7	96,111,15
9,32,60.0	41,42,10	68,85,40	96,125,30
10,11,8.0	41,57,30	69,70,8	97,98,15
10,33,60.0	42,43,10	70,71,15	97,112,23
11,14,20.0	42,58,30	70,86,40	98,99,9
12,13,25	43,44,10	71,72,15	98,113,20
12,18,15	43,59,30	72,73,15	99,100,10
13,20,20	44,45,10	73,74,10	99,114,20
14,21,15	44,60,30	73,87,40	100,101,17
14,22,15	45,46,10	74,75,10	100,115,20
15,16,15	45,61,30	74,88,40	101,102,15
15,28,25	46,47,10	75,76,10	101,116,20
16,17.0,17.0	46,62,30	75,89,40	102,103,45
16,29.0,30.0	47,48,10	76,77,10	102,130,40
17,18.0,7.0	47,63,30	76,90,40	103,104,10
17,36.0,36.0	48,49,30	77,78,10	103,117,20
18,30.0,36.0	48,64,40	77,91,40	104,105,10
19,22.0,45.0	50,53,28	78,79,10	104,118,20
19,69.0,90.0	50,55,45	78,92,40	105,106,10
20,21.0,40.0	51,52,13	78,315,32	105,119,20
20,23.0,15.0	51,65,23	79,80,10	106,107,10
21,22.0,10.0	52,66,20	79,93,40	106,120,20
21,24.0,15.0	53,54,15	80,94,45	107,108,10
22,25.0,15.0	53,69,24	81,82,14	107,121,20
22,27.0,25.0	54,55,15	81,97,35	108,109,10
23,24.0,40.0	54,70,20	82,83,10	108,122,20
23,31.0,15.0	55,56,15	82,98,30	109,110,10
24,25.0,10.0	55,71,20	83,84,10	109,123,20
25,26.0,10.0	55,72,25	83,99,30	110,111,10
25,42.0,30	56,57,15	84,85,15	110,124,20
26,43.0,30	56,72,20	84,100,30	111,125,20
27,35.0,25	57,58,10	85,86,15	112,113,8
27,45.0,25	57,73,20	85,101,30	113,114,7
28,29.0,15	58,59,10	86,87,45	113,127,20
28,51.0,50	58,74,20	86,102,30	114,115,8
31,32,15	59,60,10	87,88,10	114,128,24
31,38,10	59,75,20	87,103,30	115,128,20
32,33,15	60,61,10	88,89,10	116,129,20

117,118,10	143,160,12	172,187,30	201,202,15
117,131,20	144,145,10	173,174,15	201,209,22
118,119,10	144,161,12	173,188,30	202,203,15
118,132,20	145,146,10	174,175,15	202,210,22
119,120,10	145,162,12	174,189,30	203,204,15
119,133,20	146,147,10	175,176,8	203,211,22
120,121,10	146,163,12	176,177,8	203,204,45
120,134,20	147,148,10	176,191,35	204,205,60
120,316,9	147,164,12	177,178,15	204,215,30
121,122,10	148,149,10	177,191,30	205,207,13
121,135,20	148,165,12	178,179,15	206,207,10
122,123,10	149,150,10	178,192,30	206,221,10
122,136,20	149,166,12	179,180,10	207,208,7
123,124,10	150,151,10	179,193,30	207,222,10
123,137,20	150,167,12	180,181,10	208,223,10
124,125,10	151,152,10	180,194,30	209,210,15
124,138,20	151,168,12	181,182,10	209,224,50
125,139,22	152,170,28	181,195,30	210,211,15
126,127,15	153,154,15	182,183,10	210,224,45
126,153,50	153,171,30	182,196,30	211,212,15
127,128,15	154,155,15	183,184,10	211,225,45
127,154,50	154,172,25	183,197,30	212,213,15
128,129,15	155,156,15	184,185,10	212,226,45
128,140,25	155,173,25	184,198,30	213,214,15
128,318,40	155,318,12	185,186,10	213,227,45
129,130,15	156,157,15	185,199,30	214,215,15
129,140,20	156,174,25	186,187,12	214,228,45
130,157,50	157,158,5	186,201,15	215,216,15
131,132,10	157,175,25	187,188,15	215,229,45
131,144,40	158,159,10	187,202,15	215,230,50
132,133,10	158,176,27	188,189,15	216,217,10
132,145,40	159,160,15	188,203,15	216,230,45
133,134,10	159,177,25	189,190,15	217,218,10
133,146,40	160,161,15	189,212,35	217,231,45
134,135,10	160,178,25	190,191,15	218,219,10
134,147,40	161,162,10	190,213,35	218,232,45
134,316,11	161,179,25	191,192,15	219,220,10
135,136,10	162,163,10	191,204,15	219,233,45
135,148,40	162,180,25	191,214,35	220,221,10
136,137,10	163,164,10	192,193,15	220,234,45
136,149,40	163,181,25	192,215,35	221,222,10
137,138,10	164,165,10	193,194,10	221,235,45
137,150,40	164,182,25	193,216,35	222,223,8
138,139,10	165,166,10	194,195,10	222,319,8
138,151,40	165,183,25	194,217,35	223,237,50
138,317,12	166,167,10	195,196,10	224,225,15
139,132,42	166,184,25	195,218,35	224,238,40
140,141,20	167,168,10	196,197,10	225,226,15
140,142,40	167,169,13	196,219,35	225,226,15
140,157,33	168,170,13	197,198,10	225,239,35
140,158,35	169,170,10	197,220,35	226,227,15
141,318,15	169,185,12	198,199,10	226,240,35
141,156,12	170,200,40	198,206,25	227,228,15
142,143,15	171,172,12	199,200,8	227,241,35
142,159,12	171,186,30	199,205,12	228,229,15
143,144,15	172,173,15	200,208,25	228,242,35

229,230,15
229,243,35
230,231,15
230,244,35
230,245,40
231,232,10
232,246,35
233,234,10
233,248,35
234,235,10
234,249,35
235,236,10
235,250,35
236,237,20
236,320,15
236,321,30
237,256,40
238,239,8
238,273,70
239,240,15
240,241,15
240,257,17
241,242,15
241,257,17
242,243,15
242,265,30
243,244,16
243,258,18
244,245,5
245,246,4
246,247,3
247,248,6
247,259,15
248,249,7
249,250,7
249,263,21
250,251,7
250,279,46
251,252,7
252,253,8
252,280,45
253,254,7
253,281,44
254,255,7
255,256,7
255,260,10
256,261,12
257,268,18
258,265,16
258,269,16
259,262,6
260,264,6
260,261,9
261,267,12
262,263,8
262,270,10
264,266,6
265,268,8
265,269,16
266,271,7
266,267,13
267,272,12
268,273,22
268,274,19
269,274,9
269,270,38
270,271,60
270,275,8
271,276,8
272,276,17
272,301,52
273,282,17
274,277,8
274,275,25
275,281,30
275,278,7
276,281,14
277,278,24
277,285,16
278,279,17
278,283,9
279,280,18
279,293,22
280,281,10
280,299,35
281,287,25
282,284,15
280,281,12
280,299,50
281,287,25
282,284,15
283,286,8
284,285,30
284,289,13
284,290,18
285,286,30
285,291,10
286,287,45
286,292,8
287,288,20
287,300,20
288,294,9
289,290,15
289,296,16
290,291,18
291,292,30
291,295,13
292,293,27
292,297,18
293,294,50
293,298,16
294,300,30
294,301,25
295,297,25
296,304,20
297,298,28
297,307,40
297,308,60
298,299,16
298,303,35
298,308,55
299,300,14
299,303,30
300,302,23
301,302,25
302,303,16
303,308,40
304,305,40
304,309,18
305,306,8
306,307,12
306,310,13
307,308,30
307,310,13
307,322,10
308,312,18
309,310,30
309,313,45
309,314,70
310,311,5
310,322,16
311,312,28
311,322,20
312,314,60
313,314,20

Appendix 8: Dijkstra's Algorithm and the shortest routes

```
# AoCMM2016(2)
# encode = UTF-8

import io, collections, re, random

def create_connection_map(cg_file: io.TextIOWrapper) -> dict:
    """
    Create graph from the input value.
    :param cg_file:
    :return:
    """
    try:
        global total_points
        total_points = 0
        c_map = {}
        for line in cg_file:
            args = line.split(',')
            args = [float(i) for i in args]
            point1 = int(args[0])
            point2 = int(args[1])
            total_points = max(total_points, max(point1, point2))
            cost = args[2]
            if point1 in c_map:
                c_map[point1].append((point2, cost))
            else:
                c_map[point1] = [(point2, cost)]
            if point2 in c_map:
                c_map[point2].append((point1, cost))
            else:
                c_map[point2] = [(point1, cost)]
    except Exception as e:
        print(e)
        print('args:%r,line:%r' % (args, line))
    return c_map


def shortest_path_from_point(from_point: int) -> list:
    """
    create the shortest generated tree
    :param from_point:
    :return:
    """
    point_list = [None for i in range(total_points + 1)]
    point_list[from_point] = Point(value=0, previous_point=None, this_number=from_point)
    unsorted_point_set = set([i for i in range(1, total_points + 1)])
    sorted_point_set = set()
    while len(unsorted_point_set) > 0:
        lowest_point = None
        lowest_value = None
        for i in unsorted_point_set:
            if point_list[i] is not None and \
                    (lowest_point is None or \
                             lowest_value > point_list[i].value):
                lowest_point = i
                lowest_value = point_list[i].value

        if lowest_point is None: break
```

Appendix 9: Program for Probability Distribution

```
global latitudeEnd
global longtitudeEnd
global latitudeStart
global longtitudeStart
global comlat
global comlon
latitudeStart=40.879242
longtitudeEnd=-73.908451
latitudeEnd=40.6857
longtitudeStart=-74.020722
lat=[]#x
lon=[]#y
comlat=[]
comlon=[]
finallat=[]
finallon=[]
pointlat=[40.828236,40.827674,40.827303,40.826420,40.825263,40.824663,40.823668,40.822818,40.821635,40.820459,40.819
680,40.822110,40.820307,40.816346,40.822866,40.822015,40.818954,40.820326,40.819216,40.814149,40.815440,40.814184,4
0.814591,40.810851,40.810183,40.809550,40.807059,40.818007,40.815702,40.813517,40.812120,40.811531,40.810334,40.808
954,40.802801,40.813392,40.811788,40.810187,40.808969,40.807791,40.806410,40.805736,40.805038,40.804388,40.803706,4
0.802772,40.802017,40.801595,40.796992,40.810697,40.808851,40.808030,40.805625,40.804447,40.803237,40.802040,40.800
647,40.799988,40.799295,40.798642,40.797954,40.797020,40.796013,40.794299,40.804945,40.804173,40.802979,40.801810,4
0.801193,40.800592,40.799455,40.798253,40.796913,40.796172,40.795479,40.794821,40.794145,40.793186,40.792203,40.791
225,40.796063,40.795275,40.794660,40.794053,40.792883,40.791671,40.787898,40.787235,40.786552,40.785879,40.785200,4
0.784306,40.783274,40.782933,40.780171,40.777493,40.789932,40.788876,40.788299,40.787649,40.786504,40.785318,40.781
525,40.780875,40.780176,40.779485,40.778811,40.777869,40.776862,40.775928,40.774969,40.784984,40.784367,40.783860,4
0.783145,40.781967,40.777003,40.776337,40.775633,40.774967,40.774291,40.773308,40.772333,40.771390,40.770269,40.781
090,40.779806,40.778612,40.777426,40.776215,40.772445,40.771787,40.771113,40.770422,40.769748,40.768789,40.767798,4
0.766847,40.766043,40.773008,40.769223,40.766875,40.765680,40.764282,40.763591,40.762908,40.762225,40.761559,40.760
600,40.759617,40.758658,40.758674,40.771490,40.770304,40.769110,40.767924,40.766738,40.766324,40.765544,40.764366,4
0.762960,40.762302,40.761611,40.760904,40.760262,40.759296,40.758296,40.757353,40.755731,40.754861,40.766771,40.765
885,40.764682,40.763471,40.762289,40.761436,40.761103,40.759884,40.758559,40.757860,40.757185,40.756502,40.755827,4
0.754860,40.753877,40.762000,40.760836,40.759641,40.758402,40.757224,40.756013,40.754840,40.753481,40.752818,40.752
167,40.751461,40.750786,40.749818,40.748782,40.748067,40.748067,40.758884,40.757728,40.753621,40.746923,40.746061,4
0.745073,40.744537,40.757029,40.755769,40.754583,40.753386,40.752191,40.751010,40.749888,40.748433,40.747757,40.747
094,40.746390,40.745717,40.744775,40.743764,40.743198,40.748874,40.747704,40.746493,40.745303,40.744109,40.742905,4
0.741538,40.740879,40.740176,40.739500,40.738829,40.737886,40.736878,40.735399,40.742450,40.742109,40.740865,40.739
732,40.738563,40.737382,40.736015,40.735201,40.734786,40.734412,40.733269,40.733269,40.732338,40.731334,40.730391,4
0.729407,40.728444,40.727468,40.726775,40.736891,40.733599,40.731247,40.724864,40.724111,40.730005,40.729521,40.723
692,40.733599,40.722490,40.721537,40.733395,40.731713,40.727874,40.721275,40.718959,40.732430,40.729682,40.725928,4
0.719938,40.728378,40.725098,40.723658,40.722242,40.721467,40.729099,40.723661,40.725814,40.725548,40.722279,40.717
713,40.716277,40.724292,40.723767,40.723645,40.721043,40.719100,40.714489,40.721886,40.721650,40.717609,40.716031,4
0.714401,40.714318,40.710876,40.710396,40.709623,40.717170,40.714153,40.713062,40.712156,40.708104,40.713734,40.711
376,40.710439,40.706149,40.704885,40.702095]
pointlon=[-73.952754,-73.951315,-73.953685,-73.950490,-73.947616,-73.946217,-73.943876,-73.941898,-73.939086,-
73.936240,-73.934422,-73.948098,-73.943792,-73.934240,-73.955717,-73.953668,-73.952232,-73.949392,-73.946980,-
73.944700,-73.940001,-73.934449,-73.947863,-73.939024,-73.937395,-73.935947,-73.933674,-73.960385,-73.958285,-
73.953113,-73.949776,-73.946453,-73.943617,-73.940398,-73.930304,-73.956249,-73.954874,-73.951168,-73.948346,-
73.945503,-73.942284,-73.940642,-73.939022,-73.937423,-73.935835,-73.933625,-73.931018,-73.930063,-73.920435,-
73.954519,-73.965806,-73.963910,-73.958195,-73.955359,-73.952527,-73.949693,-73.946452,-73.944845,-73.943242,-
73.941615,-73.940034,-73.937785,-73.935414,-73.931251,-73.968508,-73.966695,-73.963863,-73.961084,-73.959646,-
73.958198,-73.955301,-73.952458,-73.949209,-73.947637,-73.946029,-73.944406,-73.942795,-73.940579,-73.938218,-
73.935978,-73.975007,-73.973215,-73.971816,-73.970360,-73.967571,-73.964688,-73.955750,-73.954166,-73.952542,-
73.950935,-73.949342,-73.947162,-73.944716,-73.943868,-73.943895,-73.942977,-73.980345,-73.977845,-73.976525,-
73.975012,-73.972180,-73.969348,-73.960440,-73.958820,-73.957211,-73.955559,-73.953960,-73.951804,-73.949390,-
73.947126,-73.944798,-73.982689,-73.981155,-73.979975,-73.978350,-73.975507,-73.963742,-73.962132,-73.960496,-
73.958881,-73.957313,-73.955060,-73.952689,-73.950447,-73.947690,-73.987210,-73.984506,-73.981631,-73.978842,-
73.975988,-73.967040,-73.965441,-73.963810,-73.962190,-73.960591,-73.958435,-73.956000,-73.953758,-73.951795,-
73.982208,-73.984761,-73.979085,-73.976242,-73.973013,-73.971447,-73.969795,-73.968186,-73.966598,-73.964377,-
```

```
73.961974,-73.959710,-73.958712,-73.994152,-73.991395,-73.988530,-73.985708,-73.982886,-73.981888,-73.980053,-
73.977199,-73.973970,-73.972393,-73.970730,-73.969121,-73.967544,-73.965281,-73.962946,-73.960682,-73.964760,-
73.962593,-73.996770,-73.994624,-73.991770,-73.988970,-73.986103,-73.984108,-73.983263,-73.980441,-73.977190,-
73.975591,-73.973982,-73.972383,-73.970784,-73.968509,-73.966138,-74.001206,-73.998310,-73.995464,-73.992644,-
73.989804,-73.986928,-73.984125,-73.980902,-73.979283,-73.977738,-73.976077,-73.974466,-73.972198,-73.969899,-
73.968156,-73.968156,-73.999712,-73.996855,-73.987133,-73.971245,-73.974954,-73.972578,-73.971215,-74.004918,-
74.001995,-73.999145,-73.996323,-73.993465,-73.990622,-73.987990,-73.984552,-73.982956,-73.981357,-73.979734,-
73.978124,-73.975884,-73.973519,-73.972110,-74.006991,-74.004180,-74.001380,-73.998489,-73.995650,-73.992804,-
73.989606,-73.987976,-73.986372,-73.984736,-73.983170,-73.980925,-73.978538,-73.974959,-74.008784,-74.008162,-
74.005169,-74.002510,-73.999668,-73.996847,-73.993625,-73.991736,-73.990788,-73.989892,-73.987199,-73.987199,-
73.984941,-73.982577,-73.980361,-73.978033,-73.975694,-73.973351,-73.971938,-74.005549,-73.999539,-73.990413,-
73.975167,-73.973378,-73.991120,-73.989913,-73.976048,-74.002845,-73.976918,-73.974647,-74.004237,-74.000978,-
73.993134,-73.977800,-73.975001,-74.010235,-74.002193,-73.994649,-73.978758,-74.002846,-73.995321,-73.991052,-
73.986287,-73.983844,-74.010696,-73.996498,-74.011078,-74.004066,-73.997103,-73.985783,-73.980674,-74.011256,-
74.007969,-74.004793,-73.998227,-73.993378,-73.981619,-74.005361,-74.011777,-74.000244,-73.995513,-73.990237,-
73.987161,-73.980542,-73.986583,-73.991964,-74.012894,-74.006296,-74.007234,-74.005596,-73.999462,-74.013849,-
74.008688,-74.009458,-74.003040,-74.015594,-74.015761]
event=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,28,181,183,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,833,1114,936,7,0,0,0,0,0,0,0,0,0,0,0,0,390,1228,8
64,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,242,1335,1227,388,0,0,0,0,0,0,0,0,0,0,0,0,0,41,1156,1023,262,0,0,0,0,0,0,0,0,0,0,0,0,0,470,1
125,984,1556,592,0,0,0,0,0,0,0,0,0,0,0,0,0,242,1272,1287,1271,1425,680,0,0,0,0,0,0,0,0,0,0,0,23,481,1022,776,1075,1261,2106,
2352,82,3,0,0,0,0,0,0,0,0,116,1040,1528,786,95,824,1654,1199,787,70,5,1,0,0,0,0,0,0,189,1608,1587,803,126,1095,2149,2090,2
94,0,7,0,0,0,0,0,0,0,1062,1372,2438,537,448,1720,2299,1852,544,24,0,0,0,0,0,0,0,246,2640,4312,3725,4088,3708,4040,5298,27
56,544,0,0,0,0,0,0,0,0,0,1511,2506,4138,4491,3954,4575,2912,704,26,0,0,0,0,0,0,0,0,0,174,2125,2221,2727,2765,3066,3295,310
,0,0,0,0,0,0,0,0,0,0,0,506,1745,1810,1696,2683,2026,764,1,0,0,0,0,0,0,0,0,0,0,23,980,3041,3833,4459,3549,1272,567,2,0,0,0,0,0,
0,0,0,0,0,834,1657,2034,1408,1298,532,369,103,7,0,0,0,0,0,0,0,0,0,0,275,819,244,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0]
result=[]
for x in range(0,20):
    lat.append((latitudeEnd-latitudeStart)/19*(x-1)+latitudeStart)
for x in range(0,20):
    lon.append((longtitudeEnd-longtitudeStart)/19*(x-1)+longtitudeStart)
def latitude(k):
    a=0
    while k< lat[a]:
        a=a+1
    return a
def longtitude(k):
    a=0
    while k< lon[a]:
        a=a+1
    return a
def find(lati,long):
    return event[lati+(long)*19]
for x in range(0,len(pointlon)):
    finallon.append(longtitude(pointlon[x]))
for x in range(0,len(pointlat)):
    finallat.append(latitude(pointlat[x]))
for x in range(0,len(pointlat)):
    result.append(find(finallat[x],finallon[x])/(1550*24*60))

#output
print(result)
#for x in range(0,len(result)):
#    print("The",x+1,"point: ")
#    print(result[x])
```

Appendix 10: Probability Output

0.0, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 0.0, 0.0, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.2544802867383513e-05, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.3440860215053763e-06, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.3440860215053763e-06, 1.2544802867383513e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 1.2544802867383513e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.198924731182796e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.10931899641577e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.10931899641577e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 8.198924731182796e-05, 0.0, 0.0,0,0,0,0,0,0,0,0

Appendix 11: Static Model and Simulations

```
#!/usr/bin/env/python3
# encode = UTF-8

import io, collections, re, random


def create_connection_map(cg_file: io.TextIOWrapper) -> dict:

    try:
        global total_points
        total_points = 0
        c_map = {}
        for line in cg_file:
            args = line.split(',')
            args = [float(i) for i in args]
            point1 = int(args[0])
            point2 = int(args[1])
            total_points = max(total_points, max(point1, point2))
            cost = args[2]
            if point1 in c_map:
                c_map[point1].append((point2, cost))
            else:
                c_map[point1] = [(point2, cost)]
            if point2 in c_map:
                c_map[point2].append((point1, cost))
            else:
                c_map[point2] = [(point1, cost)]
    except Exception as e:
        print(e)
        print('args:%r,line:%r' % (args, line))
    return c_map


def shortest_path_from_point(from_point: int) -> list:

    point_list = [None for i in range(total_points + 1)]
    point_list[from_point] = Point(value=0, previous_point=None, this_number=from_point)
    unsorted_point_set = set([i for i in range(1, total_points + 1)])
    sorted_point_set = set()
    while len(unsorted_point_set) > 0:
        lowest_point = None
        lowest_value = None
        for i in unsorted_point_set:
            if point_list[i] is not None and \
                    (lowest_point is None or \
                             lowest_value > point_list[i].value):
                lowest_point = i
                lowest_value = point_list[i].value

        if lowest_point is None: break

        for to_point, cost in connection_map.get(lowest_point):
            if point_list[to_point] is None or point_list[to_point].value > \
                        lowest_value + cost:
                point_list[to_point] = Point(value=lowest_value + cost,
                                             previous_point=point_list[lowest_point].this_number,
                                             this_number=to_point)
        unsorted_point_set.remove(lowest_point)
        sorted_point_set.add(lowest_point)
```

```
    return point_list


def nearest_hospital(hospital_list: list) -> list:

    to_which_hospital = [None]
    for i in range(1, total_points + 1):
        cost = None
        hospital = None
        for hospital_point, spanning_tree in hospital_list:
            if hospital is None or spanning_tree[i].value < cost:
                hospital = (hospital_point, spanning_tree)
                cost = spanning_tree[i].value
        to_which_hospital.append((hospital, cost))
    return to_which_hospital


def trace_tree(destination_point: int, spanning_tree: list) -> str:

    trace_point = spanning_tree[destination_point]
    return_string = '%r(%r)' % (destination_point, trace_point.value)
    while trace_point.previous_point is not None:
        trace_point = spanning_tree[trace_point.previous_point]
        return_string = '%r(%r)-> ' % (trace_point.this_number, trace_point.value) + return_string
    return return_string


def input_hospitals(line_input: bool) -> list:

    if line_input:
        hospital = input("输入医院所在点。每行输入一个，输完全部点后输入 fin 结束，输入 rm 撤销最后一次输入\n")
        hospital_list = []
        while hospital != 'fin':
            if hospital == 'rm':
                if len(hospital_list) > 0:
                    hospital_list.pop()
                    print("已撤销最后一次输入，剩余内容为：%r" % hospital_list)
                else:
                    print("Wrong")
            elif not re.fullmatch('\d+', hospital):
                print("你的输入不是数字。重新输入。")
            elif int(hospital) > total_points:
                print("医院所在点大于最大点。重新输入。")
            elif int(hospital) in hospital_list:
                print("重复输入：列表中已包含此医院。重新输入。")
            else:
                hospital_list.append(int(hospital))
            hospital = input()
    else:
        hospital_list = open_file_as_list("医院列表", int, None, 0)
        for hospital in hospital_list:
            if hospital > total_points:
                raise Exception('医院所在点大于最大点号码')
    return hospital_list


def simulator(possibility_list: list, per_event: int,
              ambulance_limit: int, time_limit: int) -> None:
```

```
    random.seed()
    simulate_time = 0
    global ambulance_table
    # 车队初始化
    ambulance_table = [[Ambulance(status='idle', timeleft=None, destination=None)
                        for i in range(ambulance_limit)]
                       for j in range(len(hospital_list))]
    # 循环部分开始
    while simulate_time <= time_limit:
        print("%r 时间运行结果:" % simulate_time)
        for i in range(len(ambulance_table)):
            for j in range(len(ambulance_table[i])):
                if ambulance_table[i][j].status != 'idle':
                    ambulance_table[i][j] = Ambulance(status='busy',
                                                      destination=ambulance_table[i][j].destination,
                                                      timeleft=ambulance_table[i][j].timeleft - per_event)
                    if len(ambulance_table[i][j].destination) >= 2 and \
                            dest_sum(ambulance_table[i][j].destination[1:]) < ambulance_table[i][j].timeleft:
                        ambulance_table[i][j] = Ambulance(status='busy',
                                                          destination=ambulance_table[i][j].destination[1:],
                                                          timeleft=ambulance_table[i][j].timeleft)
                    if ambulance_table[i][j].timeleft <= 0:
                        ambulance_table[i][j] = Ambulance(status='idle',
                                                          destination=None,
                                                          timeleft=None)

        for i in range(len(possibility_list)):
            x = random.randint(1, 100000000000000000)
            if x <= int(possibility_list[i] * 100000000000000000):
                print("{0}点发生事故".format(i))
                raise_emergency(i)
        simulate_time += per_event

def raise_emergency(place: int):

    the_hospital_ambulance_list = ambulance_table[nearest_hospital_for_point[place][0][0] - 315]
    timeleft = 9999
    ambulance_id = -1
    for i in range(len(the_hospital_ambulance_list)):
        # 先检测辖区是否有救护车空闲
        if the_hospital_ambulance_list[i].status == 'idle':
            the_hospital_ambulance_list[i] = Ambulance(status='busy',
                                                       destination=[place],
                                                       timeleft=nearest_hospital_for_point[place][1])
            print("{0}医院派出车辆前往(所在辖区）".format(nearest_hospital_for_point[place][0][0]))
            return
        else:
            if the_hospital_ambulance_list[i].timeleft <= timeleft:
                timeleft = the_hospital_ambulance_list[i].timeleft
                ambulance_id = i
    # 再从其他地方拉壮丁
    print('{0}医院救护车繁忙'.format(nearest_hospital_for_point[place][0][0]))
    for i in range(len(ambulance_table)):
        if hospital_list[i][1][place].value < timeleft:
            for j in range((len(ambulance_table[i]))):
                if ambulance_table[i][j].status == 'idle':
                    ambulance_table[i][j] = Ambulance(
                        status='busy',
                        destination=[place],
                        timeleft=hospital_list[i][1][place].value
                    )
```

```
                print('{0}医院牌车辆前往'.format(i+315))
                return
    # 如果拉不到合适的壮丁就把最近的救护车多加一个任务
    this_ambulance = ambulance_table[nearest_hospital_for_point[place][0][0] - 315][ambulance_id]
    destination = this_ambulance.destination
    destination.append(place)
    ambulance_table[nearest_hospital_for_point[place][0][0] - 315][ambulance_id] = Ambulance(
        status='busy',
        destination=destination,
        timeleft= this_ambulance.timeleft + nearest_hospital_for_point[place][1])
    print('{0}医院派本前往{1}的救护车完成任务前往
'.format(arest_hospital_for_point[place][0][0],this_ambulance.destination[:-1]))


def dest_sum(destinations: list):

    sum = 0
    for i in destinations[1:]:
        sum += nearest_hospital_for_point[i][1]
    return sum


def open_file_as_list(instruction_string: str, store_type: type,
                      process_function, first_element) -> list:

    this_list = []
    if first_element is not None: this_list.append(first_element)
    input_finish = False
    while not input_finish:
        file_name = input("%r 文件名: " % instruction_string)
        try:
            with open(file_name) as this_file:
                if process_function is not None:
                    this_list = process_function(this_file)
                for line in this_file:
                    this_list.append(store_type(line))
            input_finish = True
        except FileNotFoundError:
            print("文件名错误。（文件要跟这个程序在同一文件夹下或者输入完整路径）")
    return this_list


def main() -> None:

    global connection_map
    connection_map = open_file_as_list("graph", float, create_connection_map, None)
    global hospital_list
    hospital_list = input_hospitals(False)
    for i in range(len(hospital_list)):
        hospital_list[i] = (hospital_list[i],
                            shortest_path_from_point(hospital_list[i]))
    global nearest_hospital_for_point
    nearest_hospital_for_point = nearest_hospital(hospital_list)
    possibility_list = []
    with open('data/prob.txt') as pf:
        possibility_list = pf.readline().split(", ")
        possibility_list = [float(x) for x in possibility_list]
        possibility_list.index(0,0)
    print(len(hospital_list))
    simulator(possibility_list, 1, 6, 1000)
```

```
ambulance_table = []
hospital_list = []  # tuple (hospital id (315,316, etc.) , spanning tree)
nearest_hospital_for_point = []  # tuple ((hospital point, spanning tree), cost)
total_points = 0
connection_map = {}
Mission = collections.namedtuple('Ambulance', ['status', 'destination', 'timeleft'])
hospital_and_cost = collections.namedtuple('HaC', ['hospital_point', 'cost', 'point_number'])
Point = collections.namedtuple('Point', ['value', 'previous_point', 'this_number'])
Ambulance = collections.namedtuple('Ambulance', ['status', 'destination', 'timeleft'])
if __name__ == '__main__':
    main()
```