HW1

Member: Yuxin Zhang(yz2726), Junhan Xu(jx359)

Question 1

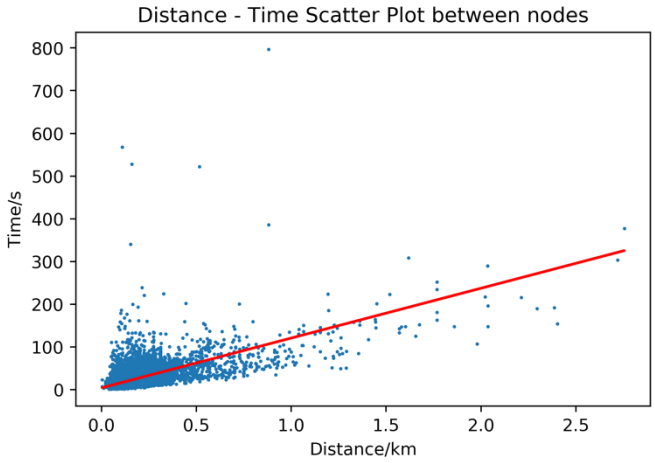
(a)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
links_data = pd.read_csv("nyc_links.csv")
node_data = pd.read_csv("nyc_nodes.csv")
# build an array of distance and add tp the links_data frame
from haversine import haversine, Unit
distance = []
link = links_data[["start",'end']].values
for i in link:
    city1 = set(node_data[node_data["name"] == i[0]][["lon","lat"]].values[0])
    city2 = set(node_data[node_data["name"] == i[1]][["lon","lat"]].values[0])
    cur_distance = haversine(city1, city2)
    distance.append(cur_distance)
links_data["Distance"] = np.array(distance)
plt.figure(dpi = 400)
plt.title("Distance - Time Scatter Plot between nodes")
plt.scatter(links_data["Distance"],links_data["time"], s =1)
plt.xlabel("Distance/km")
plt.ylabel("Time/s")
m, b = np.polyfit(links_data["Distance"], links_data["time"], 1)
plt.plot(links_data["Distance"], m*links_data["Distance"] + b, c = "r")
print("The slope of the fitted line is %s"%(m))
```

```
The slope of the fitted line is 116.6856393702052
```

After the calculation of distance, the scatter plot of distance versus time is demonstrated as the follow



The slope for the fitted straight line is 116.6856393702052

the distance calculated through haversine is in kilometers, and some of the Distance time exam ple are as follows

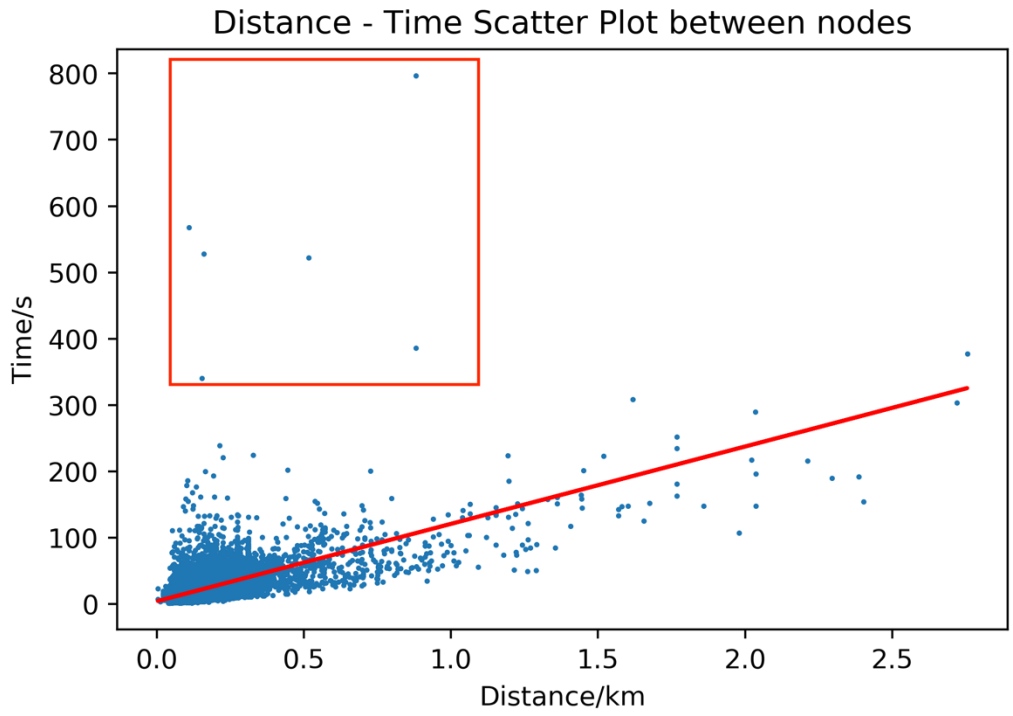
array([[0.14728633, 36.79700066],

[0.07578213, 9.84589548],

[0.07628228, 10.54777905],
[0.07628228, 10.26005846],
[0.14758088, 37.50200925],
[0.07533857, 10.527441],
[0.09706999, 16.294968],
[0.08171607, 13.25703563],
[0.07533857, 9.34319903],
[0.09706999, 16.58559667]])

0.147 kilometers, which is 147 meters, is most likely to the corresponding time of 36.8 seconds. So the unit for the time is in seconds

(b) In normal cases, the time to travel in a certain road segment would be approximately proportional to the length of the road segment (the distance), which means all points are supposed to be either on the straight line or near it. But the scatter plot shows that there are some points deviate heavily from the straight line, which can be viewed as outliers (as indicated in the photo below). There's no way to know the reasons for the strange points. They can result either from recording errors, or from those road segment being too narrow, crowded, or destructive. If it is the later case, then removing them or adjusting them might cause bias to the real world situation. Also, among the 44252 data points, less than 1% are viewed as strange, which means that even being true "outlier" or "noises", they won't affect the model we learn from the dataset too much. As a result, we should not adjust those points.



(b)

Firstly, using the distance calculate above, we can easily get the speed array (divide distance by the time of travel in each road segment). Then, we divide speed array into 5 ranges based on its value distribution percentile, so that we give same amount of road segment a certain color. We use green, blue, yellow, orange, red to indicate road segment having highest speed recorded to lowest speed recorded. And the speed range and their corresponding colors are as follows.

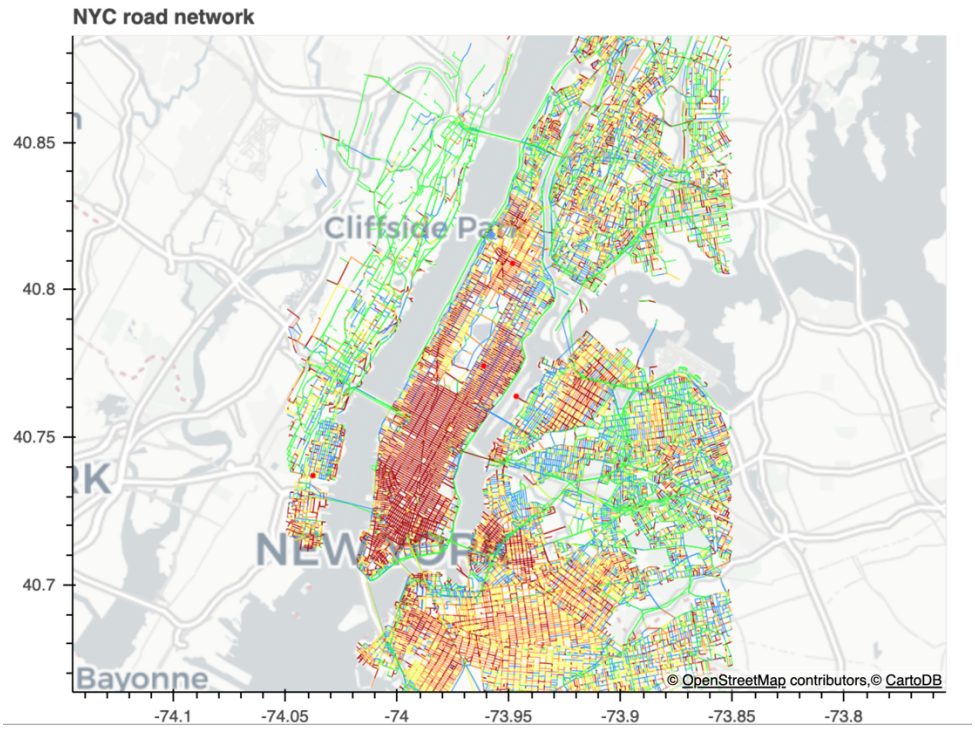
color	Speed (kilometers/second)
red	[0.0001585345941598554,0.00529218024450386)
orange	[0.00529218024450386,0.006361796886511856)
yellow	[0.006361796886511856 0.007601062214884899)
blue	[0.007601062214884899 0.010059155153690193)
green	[0.010059155153690193 0.08211845374998952)

```
import pandas as pd
import numpy as np
from bokeh.plotting import figure, show
from bokeh.tile_providers import get_provider, Vendors
from bokeh.models import (GraphRenderer, Circle, MultiLine, StaticLayoutProvider,ColumnDataSource)
def plotNetwork(nodes, links, speed,  title='Plot of Graph', target=None, on_map=False):
    dfn = nodes
    dfl = links
    e_clr = []
    # divide speed into 5 ranges based on its value (each color range as the same amount of road)
    p1 = np.percentile(np.array(speed), 20)
    p2 = np.percentile(np.array(speed), 40)
    p3 = np.percentile(np.array(speed), 60)
    p4 = np.percentile(np.array(speed), 80)

    for i in speed:
        if i>= p4:
            e_clr.append('#33FF3F')
        elif i>= p3:
            e_clr.append('#3393FF')
        elif i>= p2:
            e_clr.append('#F9FF33')
        elif i>= p1:
            e_clr.append('#FF8D33')
        else:
            e_clr.append('#A70404')
    print(min(speed),p1,p2,p3,p4,max(speed))
    # extract data
    node_ids = dfn.name.values.tolist()
    start = dfl.start.values.tolist()
    end = dfl.end.values.tolist()
    x = dfn.x.values.tolist()
    y = dfn.y.values.tolist()
    # get plot boundaries
    min_x, max_x = min(dfn.x)+2000, max(dfn.x)-2000
    min_y, max_y = min(dfn.y)+2000, max(dfn.y)-2000
    plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
                  x_axis_type="mercator", y_axis_type="mercator",
                  title=title,
                  plot_width=600, plot_height=470,
                  toolbar_location=None, tools=[]
                 )

    graph = GraphRenderer()
```

```
    if on_map == True:
        # add map tile
        plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))

    # define nodes
    graph.node_renderer.data_source.add(node_ids, 'index')
    graph.node_renderer.glyph = Circle(line_color='green', line_alpha=0,
                                       fill_color='green', size=3.5,
                                       fill_alpha=0
                                      )


    # define edges
    graph.edge_renderer.data_source.data = dict(start=list(start),
                                                end=list(end),e_clr=e_clr
                                               )
    graph.edge_renderer.glyph = MultiLine(line_color = 'e_clr',
                                          line_alpha=1, line_width=.6
                                         )

    # set node locations
    graph_layout = dict(zip(node_ids, zip(x, y)))
    graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)

    plot.renderers.append(graph)

    # add POIS
    if target != None:
        # get a small dataframe of target nodes
        dfp = dfn.loc[dfn['name'].isin(target)]
    source = ColumnDataSource(dfp)
    poi = Circle(x="x", y="y", size=3.5, line_color='white',
                 fill_color="red", line_width=0.1
                )

    plot.add_glyph(source, poi)

    show(plot)
```

```
#load nodes
data1 = pd.read_csv('nyc_nodes.csv')
dfn = pd.DataFrame(data1)
# load links
data2 = pd.read_csv('nyc_links.csv')
distance = []
link = data2[["start",'end']].values
for i in link:
    city1 = set(data1[data1["name"] == i[0]][["lon","lat"]].values[0])
    city2 = set(data1[data1["name"] == i[1]][["lon","lat"]].values[0])
    cur_distance = haversine(city1, city2)
    distance.append(cur_distance)
data2["Distance"] = np.array(distance)
speed = (data2["Distance"] / data2["time"]).values
poi = [1241986499, 42446461, 42439861,103864622]
plotNetwork(dfn,df1,speed, title="NYC road network", target=poi, on_map=True)
```

```
0.0001585345941598554 0.00529218024450386 0.006361796886511856 0.007601062214884899 0.010059155153690193 0.0821184537
4998952
```



(c) data2["ambulance_travel_time"] = 0.9 * data2["time"]

Question 2

(a) # of calls per hour = (1, 531, 870 calls a year/ 365 days a year) / 24 hours a day =174.8710046 calls across the entire city

(b) Since the probability is proportional to that of the population, the probability of a region calling equals the population percentage of that region over the entire city population.

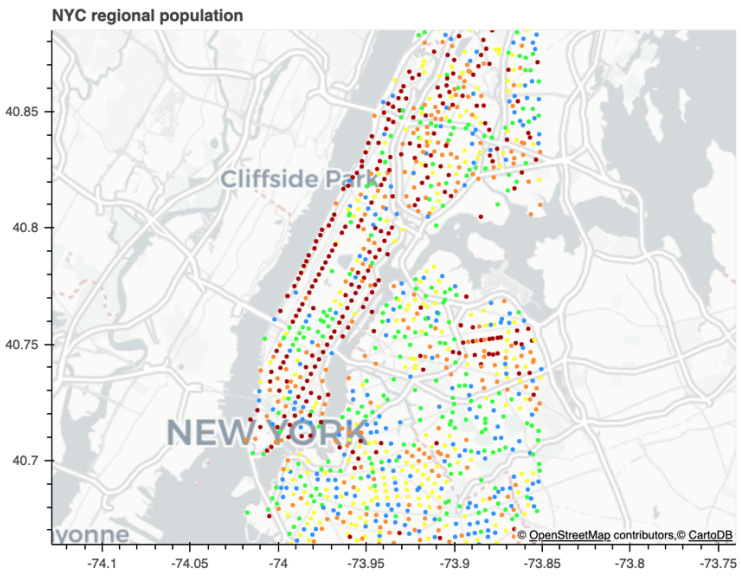
```
pop_data = pd.read_csv("nyc_population.csv")
pop_data["pop_percentage"] = pop_data["pop"] / np.sum(pop_data["pop"].values)
```

(c) We alter the previous plotting function to only include the node-plotting. We merge the population dataset with the node dataset on “name”. Then, we divide population values into 5 ranges based on its value distribution percentile, so that we give same amount of node regions to each color. We use green, blue, yellow, orange, red to indicate regions having lowest population recorded to highest population recorded. And the population ranges and their corresponding colors are as follows.

color	population
red	[2, 2391.6000000000004)
orange	[2391.6000000000004,3406.4)
yellow	[3406.4,4250.2)
blue	[4250.2,5820.200000000001)
green	[5820.200000000001,16538)

```
import pandas as pd
import numpy as np
from bokeh.plotting import figure, show
from bokeh.tile_providers import get_provider, Vendors
from bokeh.models import (GraphRenderer, Circle, MultiLine, StaticLayoutProvider,ColumnDataSource)
def plotPopulation(nodes, title='Plot of Graph', target=None, on_map=False):
    dfn = nodes
    # extract data
    node_ids = dfn.name.values.tolist()
    x = dfn.x.values.tolist()
    y = dfn.y.values.tolist()
    population = dfn["pop"].tolist()
    # get plot boundaries
    min_x, max_x = min(dfn.x)+2000, max(dfn.x)-2000
    min_y, max_y = min(dfn.y)+2000, max(dfn.y)-2000
    plot = figure(x_range=(min_x, max_x), y_range=(min_y, max_y),
                  x_axis_type="mercator", y_axis_type="mercator",
                  title=title,
                  plot_width=600, plot_height=470,
                  toolbar_location=None, tools=[]
                 )
    graph = GraphRenderer()
    # define the population color ranges, 5 ranges (each color has same amount of nodes)
    e_clr = []
    p1 = np.percentile(np.array(population), 20)
    p2 = np.percentile(np.array(population), 40)
    p3 = np.percentile(np.array(population), 60)
    p4 = np.percentile(np.array(population), 80)
    print(min(population),p1,p2,p3,p4, max(population))
    for i in population:
        if i>= p4:
            e_clr.append('#A70404')
        elif i>= p3:
            e_clr.append('#FF8D33')
        elif i>= p2:
            e_clr.append('#F9FF33')
        elif i>= p1:
            e_clr.append('#3393FF')
        else:
            e_clr.append('#33FF3F')
    if on_map == True:
        # add map tile
        plot.add_tile(get_provider(Vendors.CARTODBPOSITRON_RETINA))
    # define nodes
    graph.node_renderer.data_source.add(node_ids, 'index')
    graph.node_renderer.data_source.add(e_clr, 'color')
    graph.node_renderer.glyph = Circle(line_color='black', line_alpha=0,
                                       fill_color= "color", size=3.5,
                                       fill_alpha=1
                                      )
    # set node locations
    graph_layout = dict(zip(node_ids, zip(x, y)))
    graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)
    plot.renderers.append(graph)
    show(plot)
#load nodes
data1 = pd.read_csv('nyc_nodes.csv')
data2 = pd.read_csv("nyc_population.csv")
dfn = data2.merge(data1, left_on='name', right_on='name')
df = pd.DataFrame(dfn)
plotPopulation(df, title="NYC regional population", on_map=True)
```

```
2 2391.6000000000004 3406.4 4250.2 5820.200000000001 16538
```



(d)Firstly, we build the Graph using network package, with source and target from the edge dataset and the weight being the ambulance travel time at each edge. Then, we enumerate all possible pairs of representative points and directly pass the node names to nx.shortest_path_length to get the minimal travel time between all pair of representative points. Thus, we build a table of the shortest travel time between any two representative points.

```
# load the avatiable road segment
edge_data = pd.read_csv('nyc_links.csv')
edge_data["ambulance_time"] = 0.9 * edge_data["time"]
```

```
# the representitive points
rep_data = pd.read_csv("nyc_population.csv")
rep_points = rep_data.name.values.tolist()
```

```
import networkx as nx
edges = pd.DataFrame(
    {
        "source": edge_data["start"],
        "target": edge_data["end"],
        "weight": edge_data["ambulance_time"],
#         "color": ["red", "blue", "blue"],
    }
)
G = nx.from_pandas_edgelist(edges, edge_attr=True)
```

```
# enumerate all possible pair of representative points and calculate smallest travel time between each
start_point = []
end_point =[]
smallest_time =[]
count = 0
for i in range(len(rep_points)):
    for j in range(i,len(rep_points)):
        start_point.append(rep_points[i])
        end_point.append(rep_points[j])
        cur_time = nx.shortest_path_length(G, source=rep_points[i], target=rep_points[j],weight  ="weight")
        smallest_time.append(cur_time)
        count += 1
        if count%2000 == 0:
            print(count)
dic = {"start_point":start_point,"end_point":end_point,"smallest_time":smallest_time}
df_time = pd.DataFrame(dic)
df_time.head()
```

To attain the representative point of each location, we first searched their longitude and latitude from https://www.latlong.net/. However, the geographic location from search does not corresponding to the exact latitude and longitude presented in the node dataset. So, firstly, given some digit flexibility, we filter out some node representatives that are possible candidates for the required location and calculated their distance from the exact geographic location for the required spot, taking the one with shortest distance.
Using the representative points identified above, we can attain the shortest travel time between the required two pair of points directly from the table built above.

The travel time from Grand Central Station to the site of the World Trade Center is 943.342632 3648773 seconds
The travel time from from Cornell Tech to Columbia University is 1061.659866493137 seconds

```python
def find_closest_representative_point(exact_loc, node_data, interval):
    def distance(start_loc, end_loc):
        lat1, lon1 = start_loc
        lat2, lon2 = end_loc
        radius = 6371 # in km

        delta_lat = math.radians(lat2-lat1)
        delta_lon = math.radians(lon2-lon1)
        a = math.sin(delta_lat/2) * math.sin(delta_lat/2) + math.cos(math.radians(lat1)) \
            * math.cos(math.radians(lat2)) * math.sin(delta_lon/2) * math.sin(delta_lon/2)
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
        d = radius * c
        return d
    #filter the possible point representatives near the exact_loc
    print(exact_loc[1])
    filtered_node = node_data[(node_data["lon"] > (exact_loc[1] - interval))\
                              &(node_data["lon"] < (exact_loc[1] + interval))& \
                              (node_data["lat"] < (exact_loc[0] + interval))& \
                              (node_data["lat"] > (exact_loc[0] - interval))]
    if len(filtered_node) == 0:
        return "no available nodes in this interval"
    else:
        candidate = filtered_node[["lat","lon","name"]].values
        best_candidate = None
        shortest_distance = 100000
        for i in candidate:
            cur_distance = distance(exact_loc, set(i[:2]))
            if cur_distance < shortest_distance:
                shortest_distance = cur_distance
                best_candidate = i[2]
        return best_candidate, shortest_distance
```

```python
Grand_Central_Station = (40.752655, -73.977295)
World_Trade_Center = (40.712742,-74.013382)
Cornell_Tech = (40.807260,-73.854840)
Columbia_University = (40.804496,-73.957162)
start_name1, s_d_1 = find_closest_representative_point(Grand_Central_Station, node_data, 0.01)
end_name1, s_d_2 = find_closest_representative_point(World_Trade_Center, node_data, 0.01)
start_name2, s_d_3 = find_closest_representative_point(Cornell_Tech, node_data, 0.01)
end_name2, s_d_4 = find_closest_representative_point(Columbia_University , node_data, 0.01)
time1 = df_time[(df_time["start_point"] == start_name1)&(df_time["end_point"] == end_name1)]["smallest_time"].values[0
time2 = df_time[(df_time["start_point"] == end_name2)&(df_time["end_point"] == start_name2)]["smallest_time"].values[0
print("The travel time from Grand Central Station to the site of the World Trade Center is %s seconds"%time1)
print("The travel time from from Cornell Tech to Columbia University is %s seconds"%time2)
```

```
-73.977295
-74.013382
-73.85484
-73.957162
The travel time from Grand Central Station to the site of the World Trade Center is 943.3426323648773 seconds
The travel time from from Cornell Tech to Columbia University is 1061.659866493137 seconds
```

Q 3

Data source:

1. # of registered voters from each state: https://worldpopulationreview.com/state-rankings/number-of-registered-voters-by-state, we use the # of registered voters instead of the state population, as it being a better indicator for the actual number of population votes casted fat each state.
2. # of electoral voters for each state: https://www.britannica.com/topic/United-States-Electoral-College-Votes-by-State-1787124

Decision variable: Xi= the number of population votes (you) win in state i($Xi\ is\ an\ non-negative\ integer$)

Yi = binary variable for winning state i($Yi = 1\ if\ (you)\ win\ state\ i\ and\ Yi = 0\ otherwise$)

Objective function:

Min $\dfrac{\sum Xi}{total\ registered\ voters\ from\ all\ states}$

Constraints:

$$\frac{1}{2} - \frac{Xi}{\#\ of\ registed\ voters\ in\ state\ i} < (larger\ M)(1 - Yi)$$

$$\frac{Xi}{\#\ of\ registed\ voters\ in\ state\ i} - \frac{1}{2} < (larger\ M)Yi$$

$$\sum Yi * (\#\ of\ electoral\ votes\ in\ state\ i) \geq 270$$

Setting up the Integer Optimization problem in OR-Tool, we have

```
from ortools.linear_solver import pywraplp
def main():
    data = pd.read_csv("election_data.csv")
    register = list(data["total Registered"])
    electoral = list(data["Electoral votes"])
    # Create the linear solver with the GLOP backend.
    solver = pywraplp.Solver.CreateSolver('Example','SCIP')
    M = 1000
    # Create the variables
    myVars = [[0 for i in range(51)],[0 for i in range(51)]]
    for i in range(51):
        myVars[0][i] = solver.IntVar(0,solver.infinity(), 'X'+str(i+1))
        myVars[1][i] = solver.IntVar(0, 1, 'Y'+str(i+1))
    print('Number of variables =', solver.NumVariables())
    # Create a linear constraint,
    for i in range(51):
#     win_constraint += electoral[i] * myVars[1][i]
        solver.Add(0.5 -myVars[0][i] / register[i]-M * (1-myVars[1][i])  <= 0)
        solver.Add(-0.5 +myVars[0][i] / register[i]-M *myVars[1][i]  <= 0)
    solver.Add(solver.Sum([electoral[j] * myVars[1][j] for j in range(51)]) >= 270)
    print('Number of constraints =', solver.NumConstraints())
    # Create the objective function
    solver.Minimize(solver.Sum(myVars[0]) / sum(register))
    status = solver.Solve()
    if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.FEASIBLE:
        print('Minimal popular vote % = ', solver.Objective().Value(), '\n')
        for i in range(51):
            print(myVars[0][i].solution_value())
if __name__ == '__main__':
    main()
```

```
Number of variables = 102
Number of constraints = 103
Minimal popular vote % =  0.21544012078864355
```

The solver gives that the minimal percentage of population vote it requires in order to win the election is only 21.54%. The difference between this solution and the approximation of 25% is that by applying a better estimation of number of population votes and electoral votes of each state, the minimal estimation is pushed down. And this is because now we can aim for winning in the priority the states with higher electoral-to-population ratio.

If modeling Maine and Nebraska are taken into account, there might be possibilities to further reduce the minimal percentage of population votes if they have very high electoral-to-population ratio.

Q4

The probability of all possible game result are as follows

			Game score with probs below in red					
				0-0				
			1:0	1	0:1			
		2:0	p	1:1	1-p	0:2		
	3:0	p^2	2:1	2p(1-p)	1:2	(1-p)^2	0:3	
win1	p^3	3:1	3p^2(1-p)	2:2	3p(1-p)^2	1:3	(1-p)^3	lose1
p^4	win2	4p^3(1-p)	3:2	6p^2(1-p)^2	2:3	4p(1-p)^3	lose2	(1-p)^4
	4p^4(1-p)	win3	10p^3(1-p)^2	3:3	10p^2(1-p)^3	lose3	4p(1-p)^4	
		10p^4(1-p)^2	win4	20p^3(1-p)^3	lose 4	10p^2(1-p)^4		
			20p^4(1-p)^3		20p^3(1-p)^4			

P(Team A wins) = P(win1) + P(win2) + P(win3) + P(win4)

$$= p^4[1 + 4(1-p) + 10(1-p)^2 + 20(1-p)^3]$$

Using the function plotting toolkit, we have the plot

Firstly when there's 0 possibility for A to win a game, it's obvious that he can't win the series, which is the same for the losing case as well. Secondly, since there are up to 7 games and whoever wins (no tie) the first 4 wins the series, the possible winning cases equals the possible losing cases. So the result space for winning and losing is symmetric, this leads to the result that when winning an individual game is random (probability = 0.5) for A, winning the series is random (probability = 0.5). As we can see from the graph, the probability for A winning the game is increasing with p, which pretty makes sense that as the probability that A beats B increases, the total probability of A winning will also increase. However, the slope is not a straight line with sloping increasing first, reaching the max slope at 0.5 and then decreasing. At the earlier stage, the increase in the total win will be more obvious with the increasing in p and will be less obvious when p reaches a certain level.

X being a binomial random variable with parameters 7 and p means that X is the number of winning cases A has in 7 series of games with p being the probability of an independent winning probability. P(X>= 4) = P(X = 4) + P(X = 5) +P(X = 6) + P(X = 7) = P(A:B = 4:0) + P(A:B = 4:1) + P(A:B = 4:2) + P(P(A:B = 4:3) = P(A wins)

According to the model, the series winning probability for a team is especially sensitive to the individual game winning probability if that probability is near 0.5. Switching the home cities for games contains the overall external factors that influences the probability of a team winning an individual game, thus helping the whole series of game to be "fair" (revealing the true capacity of the two teams), especially when the two teams are quite of equal strength.