IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC) Project 1

Omar Haque

February 23, 2017

1 SOLUTION STRATEGY

I will follow a simple algorithm for simulating the traffic flow through this system. Obviously I will have to iterate through the 200 minutes, and in each minute I will follow this process.

1. Use Dijkstra's algorithm and the temporary weight matrix to find and record where each node needs to go next
2. Move all the cars to their new locations, and keep where required (70-30 split)
3. Inject 20 cars into node 13
4. Update the temporary weight matrix
5. Update the vector of maximums

In my solution, I will start with a vector of 58 0s which I will call cars_at_node. Throughout the process cars_at_node[i] will refer to the number of cars at the node with index i. I will initialise the vector with 0s, and continue with steps 1 - 6 as above.

Step 1 is self explanatory. But for the first iteration I will need to set the temporary weight matrix equal to the original weight matrix.

Step 2 requires some work. Firstly, it is not enough to iterate through cars_at_node alone, as you cannot iterate through a vector you are updating (due to overwriting), so I need to introduce a new vector of 58 0s, cars_at_node_updated.

Then, I will iterate through all the nodes in the system, and find how many cars need to stay and move using the original vector cars_at_node. I will then insert them into their appropriate positions (given by step 1) into cars_at_node_updated, being careful to cumulatively add them so previous information is not lost.
This process is the same for all nodes except node 52, where I will need a separate if statement in order to simply remove 40% of cars from the node. (Again, updating cars_at_node_updated and not cars_at_node.
After iterating through cars_at_node and updating cars_at_node_updated, I will set cars_at_node equal to cars_at_node_updated as all cars have now moved to their new location (or not, if they were in the 30% that stayed). And finally, I will set cars_at_node_updated equal to 0s ready for the next iteration.

Step 3 is easy.

Step 4 I will use the rules given in the question, namely $w_{ij} = w_{ij}^{(0)} + \epsilon \frac{c_i + c_j}{2}$.

Step 5 I will use the fact that: maximum load of node j at time i+1 = maximum(maximum load of node j at time i,number of cars at node j at time i+1)

2 Main Solution Code

The code in *solution_final.py* contains the main program which carries out the process outlined by the question, i.e modelling the process of the cars moving across the city of Rome using the rules described. I have added scripts *solution_epsilon0.py* and *solution_accident_occurs* to help answer the related questions at the end of the project.

Below are the imports used by the main program.

```
# Imports
import numpy as np
import csv
import sys
import math as ma

# This import is needed for the last question
from solution_accident_occurs import max_index_tracker_no30
```

the variable *max_index_tracker_no30* is imported from another python script, *solution_accident_occurs.py* in order to answer one of the questions. This will be discussed in detail later.

Below are the functions required by the program. The docstring's explain their use. note: I will explain the use of the function away_from_52 properly later in the document.

```
# ---------------------------------------------------------------------------------
# ---------------------     FUNCTIONS USED     ------------------------------------
# ---------------------------------------------------------------------------------

def calcWei(RX, RY, RA, RB, RV):
    """
    This function is taken from Tutorials. It calculates the weight matrix
    given information about each node in the system.
    :param RX: The x coordinates of each node in the system
    :param RY: The y coordinates of each node in the system
    :param RA: the connectivity of each node in the system
    :param RB: the connectivity of each node in the system
    :param RV: the speed limits across each edge in the system
    :return: usable weight matrix
    """

    n = len(RX)
    wei = np.zeros((n, n), dtype=float)
    m = len(RA)
    for i in range(m):
        xa = RX[RA[i] - 1]
        ya = RY[RA[i] - 1]
        xb = RX[RB[i] - 1]
        yb = RY[RB[i] - 1]
        dd = ma.sqrt((xb - xa) ** 2 + (yb - ya) ** 2)
        tt = dd / RV[i]
        wei[RA[i] - 1, RB[i] - 1] = tt
    return wei

def Dijkst(ist, isp, wei):
    """
    This Dijkstra's algorithm implementation is taken from tutorials.

    :param ist: the index of the starting node
    :param isp: the index of the node to reach
    :param wei: the assosciated weight matrix
    :return:
    """

    # exception handling (start = stop)
    if ist == isp:
```

```python
        shpath = [ist]
        return shpath

    # initialization
    N = len(wei)
    Inf = sys.maxint
    UnVisited = np.ones(N, int)
    cost = np.ones(N) * 1.e6
    par = -np.ones(N, int) * Inf

    # set the source point and get its (unvisited) neighbors
    jj = ist
    cost[jj] = 0
    UnVisited[jj] = 0
    tmp = UnVisited * wei[jj, :]
    ineigh = np.array(tmp.nonzero()).flatten()
    L = np.array(UnVisited.nonzero()).flatten().size

    # start Dijkstra algorithm
    while (L != 0):
        # step 1: update cost of unvisited neighbors,
        #         compare and (maybe) update
        for k in ineigh:
            newcost = cost[jj] + wei[jj, k]
            if (newcost < cost[k]):
                cost[k] = newcost
                par[k] = jj

        # step 2: determine minimum-cost point among UnVisited
        #         vertices and make this point the new point
        icnsdr = np.array(UnVisited.nonzero()).flatten()
        cmin, icmin = cost[icnsdr].min(0), cost[icnsdr].argmin(0)
        jj = icnsdr[icmin]

        # step 3: update "visited"-status and determine neighbors of new point
        UnVisited[jj] = 0
        tmp = UnVisited * wei[jj, :]
        ineigh = np.array(tmp.nonzero()).flatten()
        L = np.array(UnVisited.nonzero()).flatten().size

    # determine the shortest path
    shpath = [isp]
    while par[isp] != ist:
        shpath.append(par[isp])
```

```
        isp = par[isp]
    shpath.append(ist)

    return shpath[::-1]

def next_node(path):
    """ Returns the next index (after the node itself) in the path.
        If the path contains only one node, returns the node itself.
    """
    if len(path) == 1:
        return path[0]
    else:
        return path[1]


def update_weight_matrix(epsilon, c, original_weight_matrix, noNodes=58):
    """
    This function updates the weight matrix according to step 5 of the
    Project. Note the added fix - the weight matrix is not changed if
    the original entry was 0.



    :param epsilon: given in question
    :param c: the vector containing number of cars at each node
    :param original_weight_matrix: the weight matrix given by RomeEdges
    :param noNodes: number of nodes in the system
    :return: the updated weight matrix
    """
    new_weight_matrix = np.zeros((noNodes, noNodes))
    for i in range(noNodes):
        for j in range(noNodes):
            if original_weight_matrix[i, j] != float(0):
                new_weight_matrix[i, j] = original_weight_matrix[i, j] + \
                                          (epsilon * (float(c[i]) +
                                                      float(c[j]))) / float(2)
    return new_weight_matrix


def extract_data():
    """
    This function opens the RomeVertices and RomeEdges files, and creates
    global variables RomeX, RomeY, RomeA, RomeB and RomeV. These are variables
    used to create the original weight matrix.
```

```

    """
    global RomeX, RomeY, RomeA, RomeB, RomeV
    RomeX = np.empty(0, dtype=float)
    RomeY = np.empty(0, dtype=float)
    with open('./data/RomeVertices', 'r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeX = np.concatenate((RomeX, [float(row[1])]))
            RomeY = np.concatenate((RomeY, [float(row[2])]))
    file.close()
    RomeA = np.empty(0, dtype=int)
    RomeB = np.empty(0, dtype=int)
    RomeV = np.empty(0, dtype=float)
    with open('./data/RomeEdges2', 'r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeA = np.concatenate((RomeA, [int(row[0])]))
            RomeB = np.concatenate((RomeB, [int(row[1])]))
            RomeV = np.concatenate((RomeV, [float(row[2])]))
    file.close()

def away_from_52(edge):
    """
    Tells you whether a given edge is pointing completely away from
    node 52, in both the x and y directions.
    :param edge: an edge of the form [a,b]
    :return: boolean whether or not this points to or away from 52
    """

    # extract data for access to global variables
    extract_data()

    # the edge is of the form [a,b]
    a = edge[0]
    b = edge[1]

    # use RomeX and RomeY to find the coordinates for a,b and node 52.
    a_coord = [RomeX[a - 1], RomeY[a - 1]]
    b_coord = [RomeX[b - 1], RomeY[b - 1]]
    coord52 = [RomeX[51], RomeY[51]]

    # find the change in x/y from a -> b
    x_change = b_coord[0] - a_coord[0]
```

```python
    y_change = b_coord[1] - a_coord[1]

    # find the change in x/y from a -> 52
    x_changeTo52 = coord52[0] - a_coord[0]
    y_changeto52 = coord52[1] - a_coord[1]

    # if we're at 52 we're moving away from it
    if a == 52:
        return True

    # if both point in same direction, false.
    if (x_changeTo52 > 0) and (x_change > 0):
        return False
    elif (x_changeTo52 < 0) and (x_change < 0):
        return False

    # if both point in same direction, false.
    if (y_changeto52 > 0) and (y_change > 0):
        return False
    elif (y_changeto52 < 0) and (y_change < 0):
        return False

    # all other tests have passed, so must be True.
    return True
```

Now using these functions, we can execute the main program.

```python
    # ----------------------------------------------------------------
    # ------------------    Main program    ---------------------------
    # ----------------------------------------------------------------


if __name__ == '__main__':

    # Import the rome edges file
    extract_data()

    # Use the calcWei function from tutorials, along with the data set given
    # to calculate the weight matrix. Also create a copy which is the
    # temporary weight matrix.
    weight_matrix = misc.calcWei(RomeX, RomeY, RomeA, RomeB, RomeV)
    temp_wei = weight_matrix.copy()

    # Initialise minutes and number of nodes
    minutes = 200
```

```python
    total_nodes = weight_matrix.shape[0]

    # Need a vector carNumbers which stores the number of cars at each vertex
    # in the graph.
    cars_at_node = np.zeros(total_nodes, dtype=int)
    cars_at_node_updated = cars_at_node.copy()  # cars_at_node updated is simil
    max_cars_at_node = cars_at_node.copy()  # max_cars_at_node is similar

    # To find the edges utilised, we need a 58x58 matrix of
    # False's. We will set each element to True if we move
    # cars from node i to node j.
    edge_utilised = np.zeros((total_nodes, total_nodes), dtype=bool)

    # Iterate through the 200 minutes
    for i in range(minutes):

        # Apply Dijkstra's algorithm to find the fastest path to node 52 in
        # the system. Then use next_node to find the next node in the given
        # path. (step 1)
        next_nodes = [next_node(Dijkst(node, 51, temp_wei))
                      for node in range(total_nodes)]

        # Move all cars as in steps 2,3. Iterate through every node in the
        # system to do this.
        for j_node in range(total_nodes):

            if j_node == 51:
                # We remove 40% of cars from node 52.
                cars_at_node_updated[51] += int(round(cars_at_node[51] * 0.6))
            else:

                # Initialise the number of cars at node j_node.
                number_of_cars = cars_at_node[j_node]

                # Initialise the next node to move to.
                node_to_move_to = next_nodes[j_node]

                # 70% of cars will move. to keep the total conserved,
                # the amount staying is just
                # number_of_cars - amount_moving
                amount_moving = int(round(0.7 * number_of_cars))
                amount_staying = number_of_cars - amount_moving

                # We now update cars_at_node.
```

```
                cars_at_node_updated[j_node] += amount_staying
                cars_at_node_updated[node_to_move_to] += amount_moving

                if amount_moving > 0:
                    # Update edges_utilised matrix
                    edge_utilised[j_node, node_to_move_to] = True

        # Now all cars have moved where they need to, we set cars_at_node
        # to this updated vector, and empty the updated vector for the next
        # iteration.
        cars_at_node = cars_at_node_updated.copy()
        cars_at_node_updated = np.zeros(total_nodes, dtype=int)

        # For the first 180 minutes, 20 cars are injected into node 13.
        if i <= 179:
            cars_at_node[12] += 20

        # The temporary weight matrix is updated.
        temp_wei = update_weight_matrix(0.01, cars_at_node, weight_matrix)

        # We have finished an iteration.

    # Now we calculate the maximum number of cars at each node in the system
    max_cars_at_node = [max(cars_at_node[node], max_cars_at_node[node])
                        for node in range(total_nodes)]
```

3 Questions

1. **Determine for each node the maximum load (maximum number of cars) over the 200 iterations.**

   As we have already incorporated the calculation of maximums in the loop, this is easy:

```
max_index_tracker = [[node+1, max_cars_at_node[node]]
                         for node in range(total_nodes)]
```

   Thus the i'th element of this array gives [node i, maximum number of cars of node i over the 200 iterations]. The code to print is below along with the output.

```
print('max_index_tracker is')
print(max_index_tracker[0:10])
print(max_index_tracker[10:20])
```

```python
print(max_index_tracker[20:30])
print(max_index_tracker[30:40])
print(max_index_tracker[40:50])
print(max_index_tracker[50:(len(max_index_tracker)+1)])
```

Figure 3.1: output of maximum loads

```
max_index_tracker is
[[1, 1], [2, 2], [3, 0], [4, 7], [5, 0], [6, 8], [7, 8], [8, 0], [9, 16], [10, 14]]
[[11, 0], [12, 8], [13, 28], [14, 0], [15, 28], [16, 22], [17, 7], [18, 28], [19, 11], [20, 28]]
[[21, 38], [22, 11], [23, 7], [24, 24], [25, 40], [26, 23], [27, 6], [28, 10], [29, 13], [30, 32]]
[[31, 12], [32, 19], [33, 15], [34, 15], [35, 23], [36, 9], [37, 3], [38, 14], [39, 19], [40, 30]]
[[41, 30], [42, 11], [43, 31], [44, 28], [45, 4], [46, 0], [47, 0], [48, 11], [49, 0], [50, 23]]
[[51, 18], [52, 63], [53, 17], [54, 15], [55, 12], [56, 14], [57, 11], [58, 11]]
```

2. **Which are the five most congested nodes?**
   Given this array *max_index_tracker*, we can simply sort by the second argument to find the top five most congested nodes

```python
top_five = sorted(max_index_tracker,
                  key=lambda node_and_max: -1 * node_and_max[1])[:5]
print('the five most congested nodes are')
print(top_five)
```

Figure 3.2: top five most congested nodes output

```
the five most congested nodes are
[[52, 63], [25, 40], [21, 38], [30, 32], [43, 31]]
```

In other words, the top five most congested nodes (from highest to lowest) is node 52 with 63 cars, node 25 with 40 cars, node 21 with 38 cars, node 30 with 32 cars and node 43 with 31 cars.

3. **Which edges are not utilized at all? Why?**

In the main program we defined a 58x58 boolean matrix of False's, where throughout the process if cars moved from index i to index j, we made the [i,j] element of the matrix True.

Thus, we have a matrix where indices (corresponding to edges) are *True* if they are traversed by some amount of cars (> 0) in the 200 minute process.

All that is left to do is count the number of False's, making sure that we don't count a False if the edge couldn't be traversed to begin with (in the original weight matrix). This corresponds to an element of the original weight matrix being 0.

```
# create a boolean matrix that corresponds to the condition
# described above
non_utilised_edges_matrix = (weight_matrix != float(0)) \
                             & (np.logical_not(edge_utilised))

non_utilised_edges = [[i+1, j+1] for i in range(total_nodes)
                             for j in range(total_nodes)
                             if non_utilised_edges_matrix[i, j]]
```

So an element $[l, m]$ belonging to this vector *non_utilised_edges* implies that the edge $l \to m$ is not utilised.

The code to print this and the output is below.

```
print('the non utilised edges are')
print(non_utilised_edges[0:10])
print(non_utilised_edges[10:20])
print(non_utilised_edges[20:30])
print(non_utilised_edges[30:40])
print(non_utilised_edges[40:50])
print(non_utilised_edges[50:60])
print(non_utilised_edges[60:(len(non_utilised_edges)+1)])
```

Figure 3.3: list of non utilised edges output

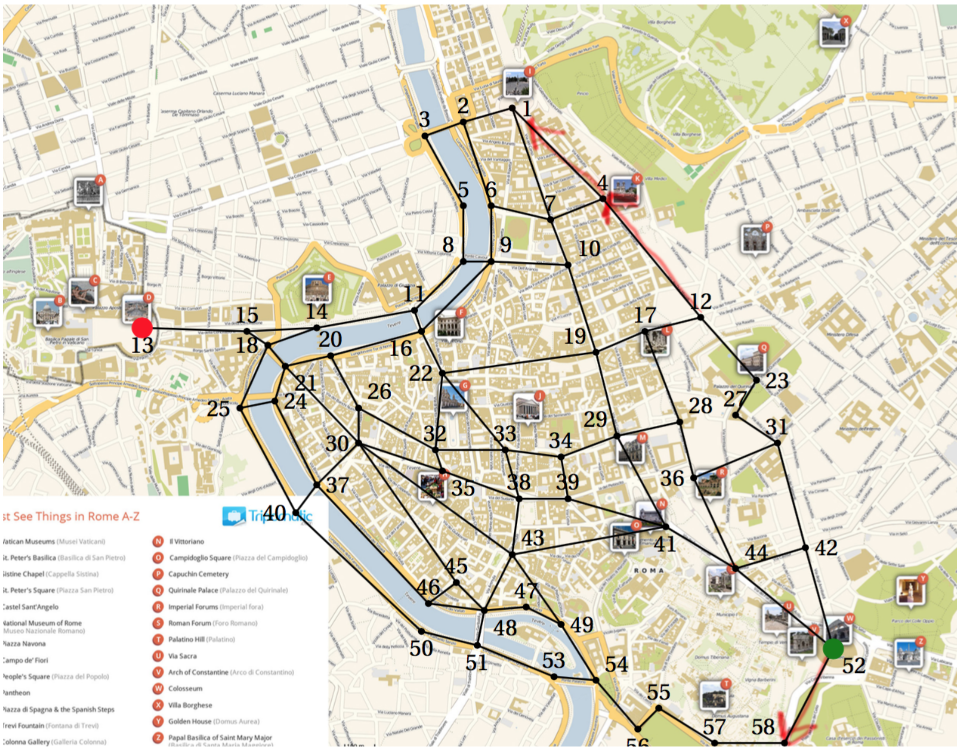
```
the non utilised edges are
[[1, 2], [1, 7], [2, 3], [3, 2], [3, 5], [4, 1], [4, 7], [5, 8], [7, 1], [7, 6]]
[[8, 9], [8, 11], [9, 8], [10, 9], [11, 14], [11, 16], [12, 4], [14, 15], [14, 18], [15, 13]]
[[16, 11], [17, 19], [18, 15], [19, 10], [21, 18], [23, 12], [27, 23], [28, 17], [28, 29], [29, 19]]
[[29, 34], [30, 21], [31, 27], [31, 36], [32, 22], [32, 26], [33, 22], [33, 32], [34, 33], [36, 28]]
[[37, 24], [38, 35], [39, 38], [41, 29], [41, 39], [41, 43], [42, 31], [42, 44], [43, 30], [43, 48]]
[[43, 49], [44, 36], [44, 41], [44, 42], [45, 30], [45, 46], [46, 37], [46, 45], [47, 48], [48, 45]]
[[48, 46], [49, 43], [49, 47], [52, 42], [52, 44], [52, 58], [54, 49], [55, 56], [56, 54], [57, 55], [58, 57]]
```

Now I answer *why* these edges aren't utilised.

For the vast majority of unused edges, it is intuitively clear that the reason they are included is because they are pointing completely away from the direction towards node 52. For example, I have highlighted edges $4 \to 1$, $12 \to 4$ and $52 \to 58$ in figure 3.4.

These edges in particular are not only in the wrong in the x direction, but wrong in the y direction too. I have created the function *away_from_52* below to return a boolean depending on whether a given edge is pointing completely away from node 52. i.e. in both x and y direction.

Figure 3.4: City of Rome graph map with drawn edges




```python
def away_from_52(edge):
    """
    Tells you whether a given edge is pointing completely away from
    node 52, in both the x and y directions.
    :param edge: an edge of the form [a,b]
    :return: boolean whether or not this points to or away from 52
    """

    # extract data for access to global variables
    extract_data()

    # the edge is of the form [a,b]
    a = edge[0]
    b = edge[1]

    # use RomeX and RomeY to find the coordinates for a,b and node 52.
    a_coord = [RomeX[a - 1], RomeY[a - 1]]
    b_coord = [RomeX[b - 1], RomeY[b - 1]]
```

```python
    coord52 = [RomeX[51], RomeY[51]]

    # find the change in x/y from a -> b
    x_change = b_coord[0] - a_coord[0]
    y_change = b_coord[1] - a_coord[1]

    # find the change in x/y from a -> 52
    x_changeTo52 = coord52[0] - a_coord[0]
    y_changeto52 = coord52[1] - a_coord[1]

    # if we're at 52 we're moving away from it
    if a == 52:
        return True

    # if both point in same direction, false.
    if (x_changeTo52 > 0) and (x_change > 0):
        return False
    elif (x_changeTo52 < 0) and (x_change < 0):
        return False

    # if both point in same direction, false.
    if (y_changeto52 > 0) and (y_change > 0):
        return False
    elif (y_changeto52 < 0) and (y_change < 0):
        return False

    # all other tests have passed, so must be True.
    return True
```

We then run this to find the number of edges facing completely away from node 52.

```python
new_unused = list(non_utilised_edges)

for _, edge in enumerate(non_utilised_edges):
        if away_from_52(edge):
        new_unused.remove(edge)

print('length of new_unused is %i' % len(new_unused))
```

Figure 3.5: length of new_unused list

So out of 71 unused edges, 41 of them were facing completely away from node 52. In a similar way we can remove nodes that are facing away from node 52 in the x direction only.

I print the remaining edges below

```
    print(new_unused[0:10])
    print(new_unused[10:20])
    print(new_unused[20:31])
```

Figure 3.6: list of unused edges that aren't facing completely away from node 52

```
[[1, 2], [1, 7], [2, 3], [3, 2], [3, 5], [4, 7], [5, 8], [8, 9], [8, 11], [11, 14]]
[[11, 16], [14, 15], [14, 18], [17, 19], [27, 23], [28, 29], [29, 34], [31, 36], [32, 22], [41, 43]]
[[42, 44], [43, 48], [43, 49], [44, 42], [45, 46], [46, 45], [47, 48], [54, 49], [56, 54], [57, 55]]
```

We note that from the first question where we found the maximum load for all cars over the 200 iterations, it is clear that nodes with maximum load 0 (i.e. node 3, 5, 8 etc) will always appear on this list since they were never visited. However, the real question is *why* they were never visited.

If we perform Dijkstra's algorithm on the original weight matrix and find the fastest route form node 13 to 52, we find this:

```
print(Dijkst(12,51,weight_matrix))
```

Figure 3.7: Dijkstra's algorithm path from 13 to 52 (python indices)

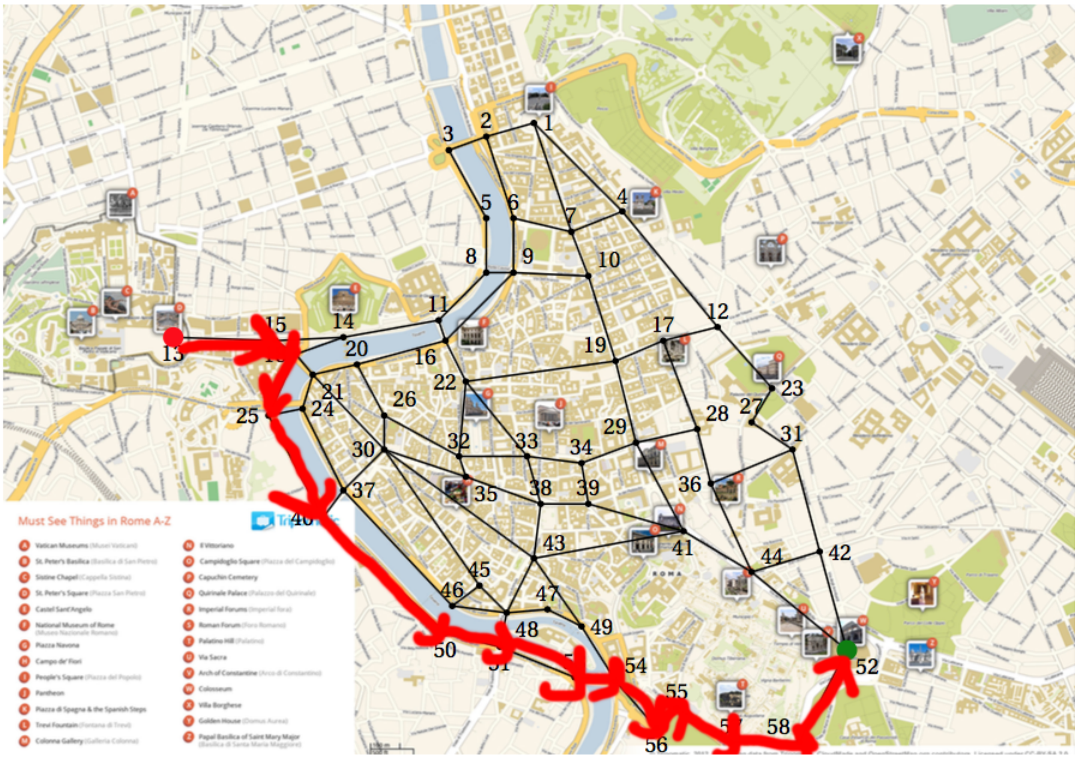
```
[12, 14, 17, 24, 39, 49, 50, 52, 53, 55, 54, 56, 57, 51]
```

note that these are the python indices and not the actual node numbers.

If we look at figure 3.8, this means without taking the effect of congestion, the fastest path throughout the city is almost completely at the bottom of the map, by the river. This means that it is unlikely for a car to need to travel through the upper west side of the city, such as through nodes 1,3,5 and 2. The only way the shortest path for a car could include such nodes is if the rest of the city is *so* congested that the shortest path will indeed require these upper left nodes. We can see that edges like [3,2], [8,9] are not optimal for this reason, and are thus unused.

There are lots of edges that although aren't facing in the wrong direction in both the x and y direction, but will obviously never lead to an optimal solution. Such as [57,55] and [56,54]. This is because these nodes have a very small degree, 2 or 3. So if the optimal path runs through them (in the opposite direction), the other direction which tracks backwards will not be part of an optimal solution.

Figure 3.8: Dijkstra's algorithm path from 13 to 52 with constant weight matrix



We can also consider the fact that if an edge is used by cars, the inverse of that edge is unlikely to be used again. This makes sense if we consider that 71 edges are unused out of a total of 156 edges (almost 1/2). So if we compare all the edges available to use and the edges unused, we should find a high correlation between pairs of the form [a,b] and [b,a] existing in either set. This is another reason highly used edges like [55,57] and [56,54] are not traversed in the opposite direction, as discussed in the paragraph above.

4. **What flow pattern do we observe for parameter $\epsilon = 0$?**

   See the *solution_epsilon0.py* file. The main program is exactly the same, but for one change. I have replaced epsilon = 0.01 with epsilon = float(0) where we update the weight matrix in the main for loop.

```python
temp_wei = update_weight_matrix(float(0), cars_at_node, weight_matrix)
```

   Now, if we consider the formula for the adjusted weight matrix at each iteration, where $w_{ij} = w_{ij}^{(0)} + \epsilon \frac{c_i + c_j}{2}$, we see that the weight matrix remains unchanged.

   As discussed in lectures, Dijkstra's algorithm is an example of a greedy algorithm. Since the time discrete process we are modelling uses Dijkstra's algorithm one node at a time (with the same weight matrix), we know that the shortest path from node 13 to node 52 will remain constant throughout the process.

Furthermore, let P be a node on the shortest path between node 13 and 52,

$$13 \rightarrow .. \rightarrow \ P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$$

Then the shortest path from node P to node 52 is

$$P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$$

Since the weight matrix is unchanged. If the shortest path between node P and node 52 were anything else then the shortest path between node 13 and node 52 would not be $13 \rightarrow ... \rightarrow P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$.

In other words, finding the shortest path from node to node with a constant weight matrix is the same as finding the overall shortest path using the same weight matrix. We can simply use the utilised edges to see that the flow of cars in the system follows this path. Printing the cars_at_node vector throughout the 200 minutes verifies this.

```python
# regular dijkstra's path
print('the Dijkstra\'s path is ')
print(dijk.Dijkst(12, 51, weight_matrix))

utilised_edges = [[i, j] for i in range(noNodes)
                  for j in range(noNodes)
                  if edge_utilised[i, j]] # this matrix is defined in the m
                                                    #code
print('the utilised edges are')
print(utilised_edges)
```

Figure 3.9: Dijkstra's path with utilised edges output (both in Python indices for clarity)

```
the Dijkstra's path is
[12, 14, 17, 24, 39, 49, 50, 52, 53, 55, 54, 56, 57, 51]
the utilised edges are
[[12, 14], [14, 17], [17, 24], [24, 39], [39, 49], [49, 50], [50, 52], [52, 53], [53, 55], [54, 56], [55, 54], [56, 57], [57, 51]]
```

Since all of these edges have a distinct head to tail connection, I can be sure the cars follow the path in the order we expect.

5. **An accident occurs at node 30 (python-index 29) which blocks any route to or from node 30. Which nodes are now the most congested and what is their maximum load? Which nodes (besides node 30) decrease the most in peak value, which nodes in- crease the most in peak value?**

   See *solution_accident_occurs.py*. The main code is exactly the same as before, but for these changes:

   Since no car can reach node 30, we need to make the 30th row and 30th column of the weight matrix equal to 0s.

```python
# Use the calcWei function from tutorials, along with the data set given
# to calculate the weight matrix. Also create a copy which is the
# temporary weight matrix.
weight_matrix = calcWei(RomeX, RomeY, RomeA, RomeB, RomeV)

# The accident at node 30 means that the 30th row and 30th column is all 0
weight_matrix[29, :] = np.zeros(58, dtype=float)
weight_matrix[:, 29] = np.zeros(58, dtype=float)

temp_wei = weight_matrix.copy()
```

Once we begin the for loop iterating through the 200 minutes, we compute the next nodes as before. But there is no path between node 30 and 52, so we add this "if node!=29" statement to the next_nodes calculation.

We also insert a 0 at index 29, so the nodes align properly again.

```python
# Iterate through the 200 minutes
for i in range(minutes):

    # Apply Dijkstra's algorithm to find the fastest path to node 52 in
    # the system. Then use next_node to find the next node in the given
    # path. (step 1)
    next_nodes = [next_node(Dijkst(node, 51, temp_wei))
                  for node in range(total_nodes) if node != 29]

    next_nodes.insert(29, 29)  # send the 0 cars from 29 to itself
```

We can ignore node 30 when moving the cars through the system, so we add this else if statement.

```python
 for j_node in range(total_nodes):

        if j_node == 51:
            # We remove 40% of cars from node 52.
            cars_at_node_updated[51] += int(round(cars_at_node[51] * 0.6))
            # really, we can just ignore node 30.
        elif j_node != 29:
```

Now we create this variable max_index_tracker_no30, which is simply the array of all the nodes with the maximum load they carry over the 200 iterations. This is then imported into the *solution_final.py* file as outlined before.

```python
# Find the top 5 most congested nodes.
max_index_tracker_no30 = [[node+1, max_cars_at_node[node]]
                          for node in range(total_nodes)]
```

Now, back in *solution_final.py* we compare and print the required values.

```python
top_eight = sorted(max_index_tracker_no30,
                   key=lambda node_and_max: -1 * node_and_max[1])[:8]
print(top_eight)
```

These are the top 8 most congested nodes when node 30 is blocked.

Figure 3.10: Top 8 most congested nodes with maximum loads when node 30 is blocked

```
[[52, 56], [25, 35], [21, 29], [13, 28], [15, 28], [18, 28], [40, 27], [20, 26]]
```

Now we find the nodes which increase/decrease the most in peak value.

```python
    differences = []
    for k in range(total_nodes):
        if k == 29:
            differences.append([k+1, 0])  # ignore when analysing
        else:
            differences.append([k+1, max_index_tracker[k][1]
                                - max_index_tracker_no30[k][1]])

    sorted_differences_most = \
        sorted(differences,
               key=lambda node_and_max: -1 * node_and_max[1])[:8]
    sorted_differences_least = \
        sorted(differences,
               key=lambda node_and_max: node_and_max[1])[:8]
    print(sorted_differences_most)
    print(sorted_differences_least)
```

Figure 3.11: The top 8 nodes which increased the most in peak value, followed by (next line) the top 8 nodes that decreased the most in peak value, all with their difference in peak values.

```
[[6, 5], [22, 4], [4, 3], [7, 3], [33, 3], [48, 3], [9, 2], [12, 2]]
[[43, -14], [21, -9], [35, -9], [41, -9], [39, -7], [52, -7], [24, -6], [44, -6]]
```

These elements represents the node number and the difference in maximum load from before to when node 30 is congested. So for example, node 6 increased its peak value by 5, and node 43 decreased its peak value by 14.