IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC) - Project 1

Isa Majothi

February 22, 2017

1 APPROACH AND SOLUTION STRATEGY

In order to model the flow of traffic through Rome in the models given to us, I defined a function (which is explained thoroughly in Section 2) called **trafficFLow** which takes as arguments numerous parameters which can be altered to simulate different traffic flows through the network. In the 3 models that we were required to simulate, the starting node is node 13 (St. Peter's Square) of python-index 12 and the end destination of the cars is node 52 (Coliseum) of python-index 51, and at each iteration some of the cars move to the next node in the current optimal path from the current node to node 52, which is determined by Dijkstra's algorithm (which was provided to us), whilst others are forced to remain behind.

The function **trafficFLow** will complete 200 iterations as instructed, each time updating the number of cars at each node as we model the cars moving through the network. I start off assuming that there are no cars in the network, so inevitably there is no movement of cars during the first iteration, but then I inject 20 cars at node 13 so at the end of the first iteration there are 20 cars at node 13 and no cars everywhere else; I then add 20 cars to node 13 at the end of the next 179 iterations (so 180 injections of 20 cars in total).

At each iteration, I see where there are currently cars and using Dijkstra's algorithm I determine where these cars should move to in that current iteration based on the shortest path, which may change at every iteration due to the fact that the weights of the edges will change as the cars traverse through the network and therefore affect the Dijkstra path. For the cars not at node 52, I compute 70% of the number of cars at these nodes and round this number to the nearest integer; I add this number to the number of cars at the next node of the current Dijkstra path, and subtract this number from the number of cars at the current node to ensure that the number of cars in the network is conserved, so that cars only leave the network via node 52. This is implemented by considering cars at node 52 separately, and as instructed I compute 40% of the number of cars at node 52 and round it to the nearest

integer, before subtracting this number from the number of cars currently there to ensure that 60% of the cars remain as described.

Once the cars have been moved through the network, I update the weight matrix according to the following formula:

$$w_{ij} = w_{ij}^{(0)} + \xi \frac{c_i + c_j}{2}$$

where $w_{ij}^{(0)}$ represents the original weight value of the edge connecting node i and node j at the start of the simulation, and c_i, c_j represent the new number of cars at nodes i and j respectively. This gives a new updated weight matrix, and is then used in the next iteration when Dijkstra's algorithm is called upon.

The number of cars is stored in an array called $carFreq$, and the $t\text{-}th$ row gives the number of cars at each of the 58 nodes at the end of iteration t, where $1 \leq t \leq 200$. From this, I determine information such as what was the maximum load of cars for each node over the 200 iterations, which is stored in the array $maximumLoad$; from this, I then find which 5 nodes were the most congested over the 200 iterations, which is stored in the array $mostCongested$ and the actual values of these maximum loads is stored in $loadCongested$. In addition to this, I also determine which edges in the network were used over the course of the simulation and return these in the list $usedUnique$, whilst $unusedUnique$ is a list consisting of the unused edges.

2 PYTHON CODE TO IMPLEMENT MODELS

Below is my python function **trafficFLow**, with comments explaining the functionality and thinking behind the approach and implementation of my function to simulate the model.

```
1  #Define a function where we can model the traffic flow easily
2  #for different parameter values
3  def trafficFlow(ist,iend,wei0,minutes,numInject,freqInject,e):
4
5      #ist = index of the starting node
6      #iend = index of the end node
7      #wei0 = weight matrix at start of the traffic flow
8      #minutes = number of iterations we simulate
9      #numInject = number of cars to inject at the end of
10     #            each iteration we are adding cars at
11     #freqInject = number of times we inject new cars into
12     #             the network
13     #e = parameter used when updating the weight matrix
14
15     #Initialise the weight matrix as the original one, but
16     #note that wei will be updated at the end of each
17     #iteration after the cars have moved through the network
18     wei = wei0.copy()
19
20     #Initialise the matrix carFreq which stores the number of
21     #cars at each node at the end of each iteration once
22     #the cars have moved and we have added more cars to the
23     #network if necessary.
24
25     #Note: the first row i.e. carFreq[0,:] is a row of zeros
26     #because I start off the simulation with no cars in the
27     #network.
28
29     #Note: numNodes is the number of nodes in the network, and
30     #can be determined by looking at the size of the weight
31     #matrix
32     numNodes = int(np.shape(wei0)[0])
33     carFreq = np.zeros((minutes,numNodes),dtype=int)
34
35     #Initialise an empty list which will update and store the
36     #edges that we use as we simulate the traffic flow
37     usedEdges = []
38
39     #Complete the iterations using a for loop, iterating
40     #over t
41     for t in range(minutes):
42
```

```
        #After the first iteration is complete i.e. once we
        #have cars in the network, update the current row of
        #zeros by the previous row so we know how many cars are
        #at each node at the start of the current iteration

        #Note: we start with no cars in the network, so at
        #the start of the first iteration with python-index
        #t=0 we have that carFreq[0,:] is a row of zeros
        if t > 0:
            carFreq[t,:] = carFreq[t-1,:]

        #Make a copy of how many cars are at each node at the
        #start of the current iteration so we can refer back
        #to it when computing how many cars (if any) need to
        #move from each node
        tempVec = carFreq[t,:].copy()

        #Loop through each of the nodes and determine the next
        #node that the cars should move to using the path
        #given by Dijkstra's algorithm, where we use the
        #current weight matrix to find the current optimal
        #Dijkstra path
        for i in range(numNodes):

            #Only apply DIjkstra's algorithm to nodes where
            #there are currently cars using the indexes of
            #these nodes and the current weight matrix,
            #because otherwise we will compute a path but
            #won't have any cars to move
            if tempVec[i] != 0:

                #Store the Dijkstra path in the list shpath so
                #that we can obtain the next node to move the
                #cars to later on
                shpath = Dijkst(i,iend,wei)

                #Assign numCars to be the current number of
                #cars at the node we are currently interested
                #in, so we can refer to it later when
                #computing how many cars to move
                numCars = tempVec[i]

                #Move cars through the network if we are not
                #at the end of the path i.e. not at node with
                #python-index iend, so that we do not lose
                #cars incorrectly
                if i != iend:

```

```
                    #If we aren't at the end node, compute 70%
                    #of the number of cars currently there to
                    #the nearest integer so these are the cars
                    #to be moved. Note that we only compute
                    #this value once and we add it to the next
                    #node and subtract it from the current
                    #node in order to conserve the number of
                    #cars in the network, so there is no net
                    #gain nor loss of cars in the network
                    numMove = int(round(numCars*0.7))

                    #Obtain the python-index of the next node
                    #for the cars to move to from the Dijkstra
                    #path
                    nextNode = shpath[1]

                    #Add the cars to be moved to the number of
                    #cars at the next node
                    carFreq[t,nextNode] = carFreq[t,nextNode]
                                                + numMove

                    #Remove these cars from the node that they
                    #were at when the iteration started
                    carFreq[t,i] = carFreq[t,i] - numMove

                    #Add the edge used by the cars moving from
                    #node i+1 (python-index i) to node
                    #nextNode+1 (python-index nextNode) to the
                    #list of used edges using append
                    usedEdges.append([i+1,nextNode+1])

                #If we are at the end node, then compute 40%
                #of the cars that are currently there to the
                #nearest integer and remove them from the
                #network by subtracting them from the end node
                else:
                    numMove = int(round(numCars*0.4))
                    carFreq[t,i] = carFreq[t,i] - numMove

        #If we haven't completed the first freqInject
        #iterations, add numInject cars into the network at
        #the starting node with python-index ist in
        #conjunction with the function argument
        if t <= (freqInject-1):
            carFreq[t,ist] = carFreq[t,ist] + numInject


```

```
        #Update the weight matrix wei for the elements that
        #are non-zero, since we only update the weights
        #corresponding to places where there are actually
        #links between the nodes. We do this using the formula
        #given to us, so we add the original weight to e
        #(argument of the function) multiplied by half of the
        #sum of the new number of cars which are there at the
        #end of the current iteration
        for i in range(numNodes):
            for j in range(numNodes):
                if wei[i,j] != 0:
                    wei[i,j] = wei0[i,j] + e*(carFreq[t,i]
                                              + carFreq[t,j])/2

    #Note: at this point, we have completed the simulation of
    #the model

    #Find the maximum number of cars at each node by finding
    #the maximum value in each column, since each column
    #contains the number of cars at each node at the end of
    #each iteration
    maximumLoad = np.amax(carFreq, axis = 0)

    #Find the 5 most congested nodes from the array
    #maximumLoad, and list them from the most congested in
    #descending order in the array mostCongested.
    #Note: I need to add 1 to the indexes since python-indexes
    #start from 0 and not 1
    indexCongested = np.argsort(maximumLoad)[-5:][::-1]
    mostCongested = indexCongested + 1

    #Find how many cars are at each of the 5 most congested
    #nodes using the indexes of these nodes, and store these
    #in the array loadCongested
    loadCongested = maximumLoad[indexCongested]

    #Find the unvisited nodes by finding where the maximum
    #load of cars over all iterations was 0 in maximumLoad,
    #meaning there were never any cars there at any point
    #in the flow model. Again, I need to add 1 to remove the
    #effect of python-indexes
    unvisitedNodes = np.where(maximumLoad == 0)[0] + 1

    #Make a list of the used edges called usedUnique, which
    #removes the repeats of the used edges in usedEdges
    usedUnique = []
    [usedUnique.append(edge) for edge in usedEdges if edge not
                                                    in usedUnique]
```

```
    #Using the edge list, which is a list of lists called
    #edgeList and will be computed from the files RomeVertices
    #and RomeEdges before using this function trafficFlow,
    #take out the edges which have been used so that we can
    #obtain the unused edges. To do this, set unusedUnique to
    #be edgeList, and then remove any edges that have been
    #used from it to obtain the unused edges in the simulation
    unusedUnique = edgeList[:]
    for edge in usedUnique:
        if edge in unusedUnique:
            unusedUnique.remove(edge)

    #If e = 0, then there is one flow path for the traffic so
    #I want this path to be returned by the function along
    #with the other bits of useful information computed by the
    #function
    if e == 0:

        #Compute the Dijkstra path
        flowPattern = Dijkst(ist,iend,wei0)

        #Add 1 to each of the python-indexes in flowPattern so
        #we obtain the nodes of the Dijkstra path and not
        #their pyhton-indexes
        for i in range(len(flowPattern)):
            flowPattern[i] = flowPattern[i] + 1

        #Return all of this data
        return carFreq,maximumLoad,mostCongested,
               loadCongested,unvisitedNodes,usedUnique,
               unusedUnique,flowPattern

    #If e != 0, just return the relevant information and no
    #path since there wouldn't be a unique path taken by all
    #of the cars injected into the network as there would be
    #so many different paths taken by different sets of cars
    else:
        return carFreq,maximumLoad,mostCongested,
               loadCongested,unvisitedNodes,usedUnique,
               unusedUnique
```

3 RESULTS FOR MODELS

For reference in the subsections below in which I report on my findings, I have included the code below which I used to simulate the 3 models of interest. Note that I used the given function **calcWei** to initialise the weight matrix using the information contained in the given data files **RomeVertices** and **RomeEdges**, and then I computed a unique edge list called *edgeList* which consists of a unique list of the set of edges in the network; I do this because within my python function **trafficFlow**, I refer to *edgeList* in order to determine the list of unused edges by removing the used edges from a list to leave behind the unused edges.

```
if __name__ == '__main__':
    import numpy as np
    import scipy as sp
    import math as ma
    import sys
    import time
    import csv

    #Initialise weight matrix using the given data files
    RomeX = np.empty(0,dtype=float)
    RomeY = np.empty(0,dtype=float)
    with open('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('RomeEdges','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()

    #Make a copy of the original weight matrix using the
    #provided function calcWei and the data from RomeVertices
    #and RomeEdges, and call this original weight matrix
    #weiOriginal
    weiOriginal = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV)


```

```
    #Using RomeA and RomeB which came from the file RomeEdges,
    #I make an edge list of all of the edges which can be
    #referred to when the traffic flow is modelled below
    #within the function.
    #Note: I will make a list of lists, so each element of the
    #list is in fact a list consisting of an edge in the
    #network
    edgeList1 = []
    for i in range(np.size(RomeA)):
        edge = []
        edge.append(RomeA[i])
        edge.append(RomeB[i])
        edgeList1.append(edge)

    #Ensure that the edge list has no repeated edges in it for
    #when it is referred to when the function trafficFlow is
    #called upon below
    edgeList = []
    [edgeList.append(x) for x in edgeList1 if x not in
                                                    edgeList]
    #Make a copy of the original weight matrix to use in the
    #first 2 models
    wei0 = weiOriginal.copy()

    #Model 1: start at node 13 (python-index 12) and end at
    #node 52 (python-index 51) with e = 0.01 and no blocked
    #nodes
    carFreq1,maximumLoad1,mostCongested1,loadCongested1,
    unvisitedNodes1,usedUnique1,unusedUnique1 =
    trafficFlow(12,51,wei0,200,20,180,0.01)

    #Model 2: same as Model 1, but for e = 0 instead
    carFreq2,maximumLoad2,mostCongested2,loadCongested2,
    unvisitedNodes2,usedUnique2,unusedUnique2,flowPattern1 =
    trafficFlow(12,51,wei0,200,20,180,0)

    #Get the weight matrix for Model 3 where node 30 is
    #blocked, so that any edges linked to node 30
    #(python-index 29) are blocked off so we do not use any of
    #the edges connected to node 30, and call it wei1
    wei1 = weiOriginal.copy()
    wei1[29,:] = 0
    wei1[:,29] = 0

    #Model 3: same as Model 1, but with node 30 blocked
    carFreq3,maximumLoad3,mostCongested3,loadCongested3,
    unvisitedNodes3,usedUnique3,unusedUnique3 =
    trafficFlow(12,51,wei1,200,20,180,0.01)
```

3.1 MODEL 1: $\xi = 0.01$ AND NO BLOCKED NODES

The results for this simulation can be found in my python code, but I will reference to the necessary output when analysing the model now. The maximum load for each node over the 200 iterations was returned and stored in the array $maximumLoad1$, and this was found to be:

```
maximumLoad1
= array([   1,  2,  0,  7,  0,  8,  8,  0, 16, 14,  0, 8,  28,
            0, 28, 22,  7, 28, 11, 28, 38, 11,  7, 24, 40, 23,
            6, 10, 13, 32, 12, 19, 15, 15, 23,  9,  3, 14, 19,
           30, 30, 11, 31, 28,  4,  0,  0, 11,  0, 23, 18, 63,
           17, 15, 12, 14, 11, 11  ])
```

From this, I was able to determine the 5 most congested nodes, and these were returned and stored in the array $mostCongested1$:

```
mostCongested1 = array([52, 25, 21, 30, 43], dtype=int64)
```

so I can conclude that, in descending order, the most congested nodes were 52, 25, 21, 30 and 43. The maximum loads at each of these nodes was stored in $loadCongested1$:

```
loadCongested1 = array([63, 40, 38, 32, 31])
```

It is worth noting that after having analysed the map of Rome provided, and where these 5 nodes are in relation to the network as a whole, it is not surprising that these nodes happened to be the 5 most congested nodes. This is because node 52 is obviously the desired end destination of all of the cars, and therefore all of the cars are aiming to get to node 52. Node 25 lies near node 13 and is is the starting point of a one-way system (given we start at node 13) that leads directly to node 52 that avoids much of the city, which could explain why it was a very congested node. Nodes 21, 30 and 43 all lie pretty centrally in the network and are very well-connected to many other nodes in terms of the number of edges emanating from them, hence they provide focal points from which the cars can take many routes to node 52 and possibly explains why they were amongst the most congested nodes.

The unused edges were stored in unusedUnique1 as follows:

```
unusedUnique1
= [  [1, 2],   [2, 3],   [3, 2],   [3, 5],   [5, 8],  [8, 11],
   [11, 14], [14, 18], [14, 15], [18, 15], [15, 13], [56, 54],
   [55, 56], [57, 55], [58, 57], [52, 58], [54, 49], [49, 47],
   [47, 48], [48, 46], [46, 37], [37, 24],   [8, 9],   [9, 8],
   [11, 16], [16, 11], [21, 18],   [4, 1],  [12, 4], [23, 12],
   [27, 23], [31, 27], [42, 31], [52, 42],   [1, 7],   [7, 1],
   [19, 10], [29, 19], [41, 29], [44, 41], [52, 44], [17, 19],
   [28, 17], [36, 28], [44, 36], [44, 42], [42, 44], [31, 36],
   [28, 29],   [4, 7],   [7, 6],  [10, 9], [32, 22], [33, 22],
```

```
11     [41, 43], [43, 49], [49, 43], [32, 26], [33, 32], [34, 33],
12     [29, 34], [38, 35], [39, 38], [41, 39], [43, 30], [43, 48],
13     [48, 45], [46, 45], [45, 46], [45, 30], [30, 21]  ]
```

Also note the unvisited nodes for this model, stored in $unvisitedNodes1$:

```
1 unvisitedNodes1
2 = array([3,  5,  8, 11, 14, 46, 47, 49],dtype=int64)
```

After examining the unused edges in this model, and also observing which nodes were not visited, it appears that the unused edges mostly either lie in a one-way system or simply would be taking us backwards and away from the end destination of node 52. Taking a closer look at the list of edges, $edgeList$, reveals that there are one-way systems (running along either side of the river) which form the following paths between the nodes going from left to right:

- $[3-5-8-11-14-18-25-40-50-51-53-54]$
- $[3-5-8-11-14-15]$
- $[54-49-47-48-46-37-24-21-20-16-9-6-2]$

Comparing the unvisited nodes to the edges that appear in the one-way systems and the list of unused edges, namely $unusedUnique1$, we observe that there is clearly a correlation; if we disregard the edges that in effect take us backwards in the sense that they don't take a car towards node 52 but rather away from it, all of the other unused edges with the exception of [44 - 42] and [42 - 44] are in the one-way systems that do not take us towards node 52 given that we start at node 13, so we can conclude that the unused edges are mainly those that lie on one-way systems and do not take us towards node 52 given that we started at node 13. Also note that all of the unvisited nodes lie on one-way systems which take a car away from node 52, and every other node was used in some way.

3.2 Model 2: $\xi = 0$ and no blocked nodes

Setting the parameter $\xi = 0$ in the formula used to update the weights in effect results in a non-changing weight matrix. Because of this, the accumulation of cars at a particular node has no effect on the shortest path to node 52 at any given time, hence all of the cars follow the same path from node 13 to node 52 which is the first Dijkstra path. This flow pattern, which I called $flowPattern1$, was found to be:

```
1 flowPattern1
2 = [13, 15, 18, 25, 40, 50, 51, 53, 54, 56, 55, 57, 58, 52]
```

Observe that this path is part of a one-way system that is on the outskirts of Rome, so it makes sense that this would be the shortest path since it is a very direct path from node 13 to node 52.

3.3 MODEL 3: $\xi = 0.01$ AND NODE 30 BLOCKED

In order to simulate this model after an accident occurred at node 30, I had to modify the weight matrix which was inputted into the function **flowPattern** to account for this. As can be seen in the code at the start of this section, I set the 30^{th} row and column (with python-index 29) to 0 so that no cars could ever go through node 30 when making the journey to node 52. Since node 30 is fairly central to the network, and was the 4^{th} most congested node in Model 1, one would expect removing node 30 to have an effect on the flow of the cars through the network as we modelled the simulation.

In this model, the most congested nodes, stored in $mostCongested3$, were:

```
mostCongested3 = array([52, 25, 21, 15, 18], dtype=int64)
```

so clearly the first 3 most congested nodes haven't changed compared with Model 1 but we see that nodes 15 and 18 now become relatively more congested after blocking node 30. The maximum loads of these most congested nodes was stored in $loadCongested3$:

```
loadCongested3 = array([56, 35, 29, 28, 28])
```

so we do observe a decrease in the peak values at nodes 52, 25 and 21 of 7, 5 and 9 respectively.

In order to be able to observe which nodes decreased or increased most in peak value, I computed the difference between $maximumLoad1$ and $maximumLoad3$ as follows:

```
maximumLoad3 - maximumLoad1
= array([  1,   1,   0,   3,   0,   5,   3,   0,   2,   1,
           0,   2,   0,   0,   0,   0,   1,   0,   1,  -2,
          -9,   4,   2,  -6,  -5,  -4,   2,   1,   0, -32,
          -1,   1,   3,  -1,  -9,   1,  -3,   0,  -7,  -3,
          -9,  -2, -14,  -6,  -4,   0,   0,   3,   0,  -2,
           1,  -7,  -2,  -1,  -1,  -2,  -1,  -2  ])
```

There was inevitably a massive decrease (of 32) in the maximum load at node 30 since it was blocked, but other nodes that noticeably decreased in peak value in descending order were nodes 43, 21, 35, and 41 with a range of decrease from 9 to 14, all of which are directly connected to node 30 or in the case of node 41 connected to node 30 via node 43; we would expect there to be a reduction in the peak value of cars at the nodes closely connected to node 30 if node 30 is blocked. Nodes that increased noticeably in peak value in descending order were nodes 6, 22, 7, 33 and 48, but not by that great an amount as the range of increase in the maximum loads at these nodes was between 3 and 5.