IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

# Scientific Computing (M3SC)

John Norrie CID:01289535

February 23, 2017

## 1 THE TASK

We were given the problem of modelling the traffic flow through Rome during rush hour. There were a few basic rules for us to follow in order to achieve this.

- We build our model in discrete time($t$), where every time step represents a minute
- Each car makes it's choice of direction dependant on an local optimum route, which updates every minute.
- We inject cars at node 13(St. Peter's Basilica) and thier destination is node 52(The Coliseum).
- During each time step, the cars are moved thusly:
  - 70% of cars at each node move to the next node in the optimum path, meaning that 30% are left behind.
  - At node 52, 40% of cars leave the system, meaning 60% are left behind.
- After all cars have moved, we then update the weights of the edges using the formula $w_{ij} = {w_{ij}}^{(0)} + \xi\frac{c_i+c_j}{2}$, where ${w_{ij}}^{(0)}$ is the original weight matrix. This is then used to update the optimum path.
- We do this for 200 timesteps,the last 20 of which we do not inject anymore cars.

After creating and testing the model, I will then modify it to simulate certain scenarios, and gain more insight into the problem at hand.

## 2 Implementation of Car Movement

To implement how the cars move, intially I created a car list of length 58(I.e. each entry $c_i$ shows the number of cars at the intersection $i+1$).

```
n=len(RomeX)

c=np.zeros(n, dtype=int)
#start c[i]=0 where c[i]=number of cars
# at node i+1
```

As you can see, at the start there are no cars at any of the nodes. To begin injecting cars into the system, I created a function that injects 20 at node 13:

```
def caradd(c): #Adds new cars to the system at node 13
c[12] += 20
return c
```

After this I created a separate function that finds the optimal paths(Using the Dijkstra's Algorithm from lectures) and moves the cars about. A particular problem I encountered is that one needed to be careful not to change thier original car values when iterating over the consecutive nodes. Otherwise, cars could end up being moved more than once in the same time step. I avoided this by creating a second list *cnew* with the some dimensions as *c*, our car vector, and updating that instead. The code by which I move cars is show below:

```
def cars(L, c, cmx, wei, used):
        #Uses c=number of cars at each node
        cnew=np.zeros(L, dtype=int)
        #Creates 'dummy' vector so we don't change the
        # c[i] which running through for loop.
        #If we don't have this cars will move more than one
        # node per time step.
        for i in range(L):
                shpath = Dijkst(i,51, wei)
                #run Dijkstra's to find optimal route for each
                #node
                        if len(shpath)==1 :
                                cnew[i]+= c[i]- c[i]*4/10
                                #Cars leaving system
                        else:
                                cnew[shpath[1]] += c[i] -\
                                c[i]*3/10
                                cnew[i] += c[i]*3/10
                        #Moving 70\% of cars to next node
#Not using c[i]*7/10, as integar division truncates
#instead of rounding. If this was not done, the
```

```
22 #number of cars would not be conserved.
23 ############Changing used matrix##########
24                                     if c[i]!=0:
25                                             used[i][shpath[1]] = \
26                                              False
27                                             used[shpath[1]][i] = \
28                                              False
29 #If c[i]=0, that implies we will move cars to shpath[1]
30 #and therefore the road connecting the two will be used
31 #This will be ellaborated on later.
32 ########################################
33         cmax= np.maximum(cmx, cnew)
34         #recalulate max
35         return cnew, cmax, used
36         #puts out new, cmax, and used
```

Another point of caution when implementing is that we need the number of cars to be conserved when moving. How I acheived this can be seen in line 15 above. If instead I had used the code:

```
1 cnew[shpath[1]] += c[i]*7/10
2 cnew[i] += c[i]*3/10
```

Then the number of cars would not have been conserved during movement. This due to the fact that when integer division is performed in python, the number is not rounded, but truncated.
Note that I am also keeping track of the maximum number of cars at each vertex by comparing the current number of cars at a node($cnew_i$), and the overall maximum($cmx_i$).

## 3 Implementing Weight Updating

We require the weights of the edges to update per timestep, using the following formula: $w_{ij} = {w_{ij}}^{(0)} + \xi\frac{c_i+c_j}{2}$ where ${w_{ij}}^{(0)}$ is the original, car-less weight matrix. I acheived this by creating a square matrix, with each row = $c$. Finding the transpose of this matrix, and adding the two together gives me a matrix for which $M_{ij} = c_i + c_j$. From this we can create the new weight matrix easily:

```
1 def UpdateWei(wei0, c, z, truefalse):
2         List = [c  for i in c]
3         #Making a 'matrix' M with each row= c
4         Mat= np.array(List)
5         MatT= np.transpose(Mat)
6         wei= (z*.5*(Mat+MatT)+wei0)*truefalse
7         #0.5(M+M^T)[i][j]=(c[i]+c[j])/2 as required
8         #add original weight(wei0).
```

```
        #Change weight of edges that don't exist to zero
        #(truefalse)
        return wei
```

The truefalse matrix stops us from updating weights for edges that don't exist. truefalse is defined as such:

```
truefalse = wei0 > 0
#Matrix of boolean statements, used in weight updates.
```

I.e if $truefalse_{ij} \neq 0 \Rightarrow wei_{ij}{}^{(0)} \neq 0$ and so the edge exists. Otherwise $truefalse_{ij} = 0$. Since both truefalse and wei(our new weight) are numpy arrays, one can multiply them together and the operation will be carried out element wise(As appose to matrix multiplication for example). So by carrying this out I can set the weights for edges that don't exist to 0.

# 4 Time Steps

We want to simulate our model for 200 timesteps, not adding cars for the last 20 timesteps:

```
def flow(t=199, xi=0.01, TakeOut=-1):
#'Remove' node x from network by TakeOut=x.
#Otherwise set TakeOut=-1
        if TakeOut >= 0:
                for i in range(n):
                        wei0[TakeOut-1][i]=900
                        wei0[i][TakeOut-1]=900
                #Removing all edges to and from this node
                #This will be ellaborated on later.
        c=np.zeros(n, dtype=int)
        cmax=np.zeros(n, dtype=int)
        truefalse = wei0 > 0
        used = wei0 > 0
        wei = UpdateWei(wei0, c, xi,truefalse)

        for i in range(t):
                if i<=179:
                        c=caradd(c)
                        c, cmax, used = \
                        cars(n, c, cmax, wei, used)
                else:
                        c, cmax, used = \
                        cars(n, c, cmax, wei, used)
                wei = UpdateWei(wei0, c, xi,truefalse)
        return cmax ,used
```

In conlcusion, for each $t <= 180$, first I add cars, then I calculate an optimal route and move the cars, then I recalculate the weights. For $t > 180$ I just move the cars and update the weights. I was considering whether it would be more logical to update edges after more cars are added, and then again after they have moved. However in practice we have the same outcome, as the only edges that will be updated are those leaving node 13, and they will all be updated by the same amount, which inturn does not affect the cars' choices.

# 5 Maximum Congestion

As I have shown above, I kept track of the maximum number of cars at each node for the system. Figure 5.1 shows a plot of these maxima at thier respective intersection.

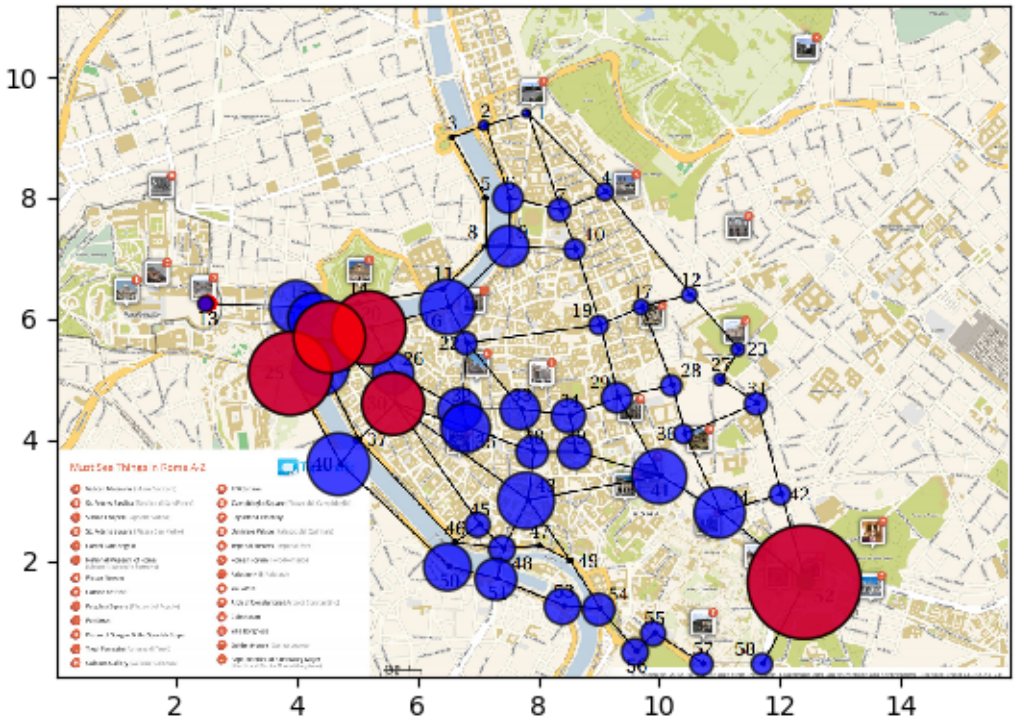


Figure 5.1: The radii of the circles about each intersection are linearly dependent to the maximum load on said intersection. The red circles represent the 5 biggest maximum values recorded.

The biggest blockage occurs when crossing the river Tiber, where the cars are forced onto a limited number of bridges. After which they pass throug Rome until the Coliseum (node

52), where there is another bottleneck as the cars leave the city.
Note that many of the nodes to the North of the city are not frequently used. This could be because any path using these nodes is less direct than that of the nodes in the centre and south, and so traveling via these nodes would increase the distance travelled dramatically. So much so, that for the most part cars prefer to go through the busy central and sourthen routes.
The five most congested nodes are, in order of size:
53 (Maximum of 58 cars), 25 (Maximum of 44 cars)
20 (Maximum of 39 cars), 21 (Maximum of 38 cars)
30 (Maximum of 33 cars)
Again three of the most congested nodes(25,20,21) are situated on the Tiber, near node 13(St. Peter's Basilicsa) implying that this is a bottle-neck point.
The code for calculating the maximum edges and plotting the graphs is shown below:

```
def plot(cmax):
                sort_index = np.argsort(cmax)[-5:]
                busy = [[i+1,cmax[i]]  for i in sort_index]
                busy = busy[::-1]
                print busy
                #Compiling and printing the 5 busiest nodes.
                a= .5*(cmax)**2 + 1
                plt.scatter(RomeX, RomeY, s=a, color='blue'\
                , edgecolors='black' , alpha=.75 ,zorder=2)
                #Creating scatterplot,
                #with varying circle radii
                Xbusy =[RomeX[i[0]-1] for i in busy]
                Ybusy =[RomeY[i[0]-1] for i in busy]
                abusy =[a[i[0]-1] for i in busy]
                plt.scatter(Xbusy,Ybusy,s=abusy, color='red'\
                , edgecolors='black' , alpha=.75,zorder=3)
                #Creating red overlay for scatter plot,
                #5 greatest maxima only.
                pic =spm.imread('Rome.png')
                plt.imshow(pic, zorder=1, \
                extent=(0.04,15.8,.1,11.17))
                #Importing and setting background.
                plt.show()

        cmax, used=flow()
        plot(cmax)
```

# 6 Non-utilized Edges

I calculate which edges have not been used by first creating a boolean array identical to the truefalse matrix from before:

```
        used = wei0 > 0
```

Then in the car-moving function($cars$), if a node is populated($c_i > 0$) then I set $used_{ij} = 0$ and $used_{ji} = 0$ where $i$ is the node we are investigating and $j$ is the next node in the optimum route. The logic behind this is that if a node(i) is populated, then 70% of those cars will move to the next node in the route(j), and so the road i↔ j has been used. The implementation is shown below:

```
        if c[i]!=0:
                used[i][shpath[1]] = False
                used[shpath[1]][i] = False
        #If c[i]=0, that implies we will move cars to
        #shpath[1], and therefore the road connecting
        #the two will be used
```

If a road $i \leftrightarrow j$ is not utilised then either $used_{ij}$ or $used_{ji}$(or both) will be non-zero. Interestingly, if only one of $used_i j$ is non-zero, then this implies that the road is a one-way street, in the direction $i \rightarrow j$. If both are non-zero, then this implies that the road is a two way street. My code for calculating these are:

```
def roads(used):
        for k in range(n):
                for j in range(k):
                #Only loop over top traingle of matrix,
                #so we do not repeat edges
                        if used[j][k]!=0 and used[k][j]!=0:
                        #This is true for a two-way road
                        #that hasn't be used
                                print '(' +str(j+1)+'<=>'+ \
                                 str(k+1)+')'
                        elif used[j][k]!=0:
                        #True for one way j->k
                                print '(' +str(j+1)+'-->'+ \
                                 str(k+1)+')'
                        elif used[k][j]!=0:
                        #True for one way k->j
                                print '(' +str(k+1)+'-->'+ \
                                 str(j+1)+')'

cmax, used=flow()
roads(used)
```

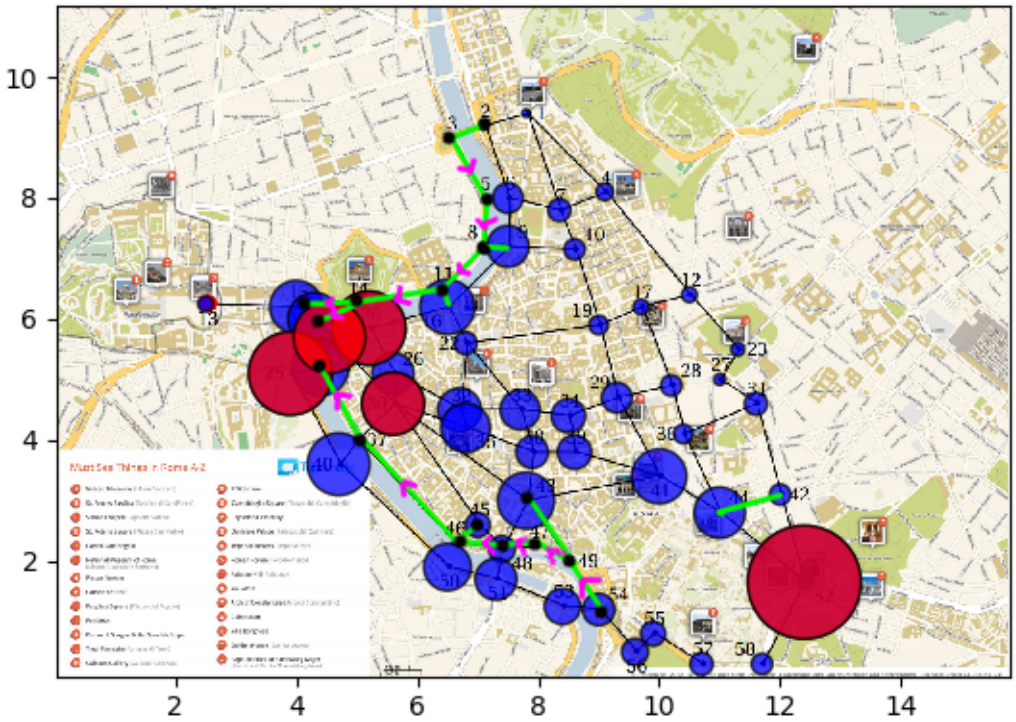


Figure 6.1: The roads that are not used are highlighted in green. The purple arrows represent the direction of the one way edges. If no arrow is present on a edge, it means it is two-way.

(2<=>3) (3–>5) (5–>8)
(8<=>9) (8–>11) (11–>14)
(14–>15) (11<=>16) (14–>18)
(37–>24) (42<=>44) (46–>37)
(45<=>46) (48–>46) (47–>48)
(43<=>49) (49–>47) (54–>49)

As the data and figure 6.1 show, many of these roads are one way roads, and the allowable direction for these roads is the opposite direction to the end point. Meaning that a car would have to go past this path, in order to use it. All of the 2 way roads which are unused (except 44<=>42) are connected to these one way roads and are also not used because of this fact.

44<=>42 is the exception. Both of these nodes are connected to the end point (Node 53). Unless the weights of the roads 44→ 53 and 42→53 are drastically different(e.i one is very large and the other very smaller), cars will always prefer just to go straight to the end point.

# 7 SETTING $\xi = 0$

In theory, setting $\xi = 0$ is equivelant to not updating the weights. Physically, this is equivelant to making the assumption that car build up does not cause congestion. A similar idea to having an inviscid flow assumption in a fluid, in comparison to a viscous flow. The particles(cars in this case) do not interact and slow eachother down. Therefore all the cars will follow one route(the optimal route of our unaltered weight matrix $wei^{(0)}$). The code for simulating this, and the figure illustrating it are below.

```
1    cmax, used=flow(xi=0)
2    plot(cmax)
```

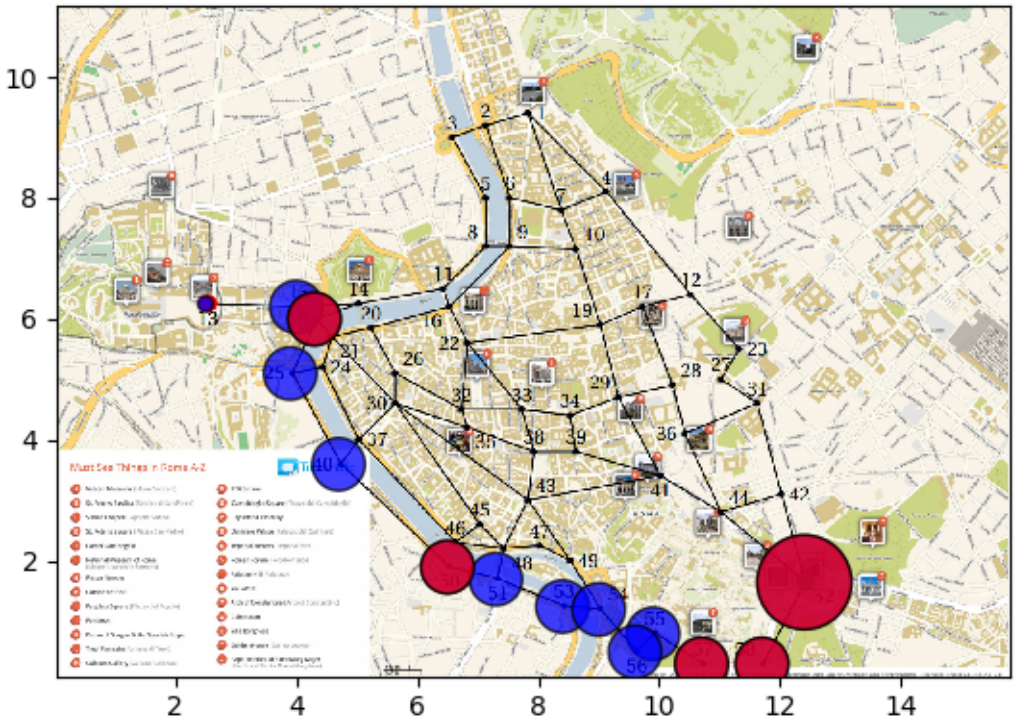


Figure 7.1: Map of car maxima when we do not update weights

The cars follow a singal route as predicted(Figure 7.1). The route is south of the river, crossing at the last bridge. Upon inspection I found that the maximum cars at node 13 = 8, at node 52 = 48, and all other used nodes have maxima = 28. In the steady state, we would expect the number of cars coming into the system(20)to be equal to the number coming

out. Under the algorithm and with the proportions of cars we move, these maxima allow this to happen continously. And so the system is attracted to this steady state.

# 8 Accident of Node 30

Now we wish to simulate the traffic flow when there is an accident at node 30, and therefore node 30 is no longer accessible. I did this by giving any edge connected to node 30 a very high initial weight. So high that when we run Dijkstra, cars will never choose to use these nodes. In practice setting these to a weight of 900 was sufficiently high to acheive this. My implementation is shows below(This is part of the function 'flow()' and can be seen is previous code displayed):

```
        TakeOut=30
        #state node you would like to 'remove' from the system
        #if not set TakeOut=-1
        if TakeOut >= 0:
                for i in range(n):
                wei0[TakeOut-1][i]=900
                wei0[i][TakeOut-1]=900
            #Add big weights to edges connected to 'removed'
            #/node
```

The outputed maxima are shown in figure 8.1.

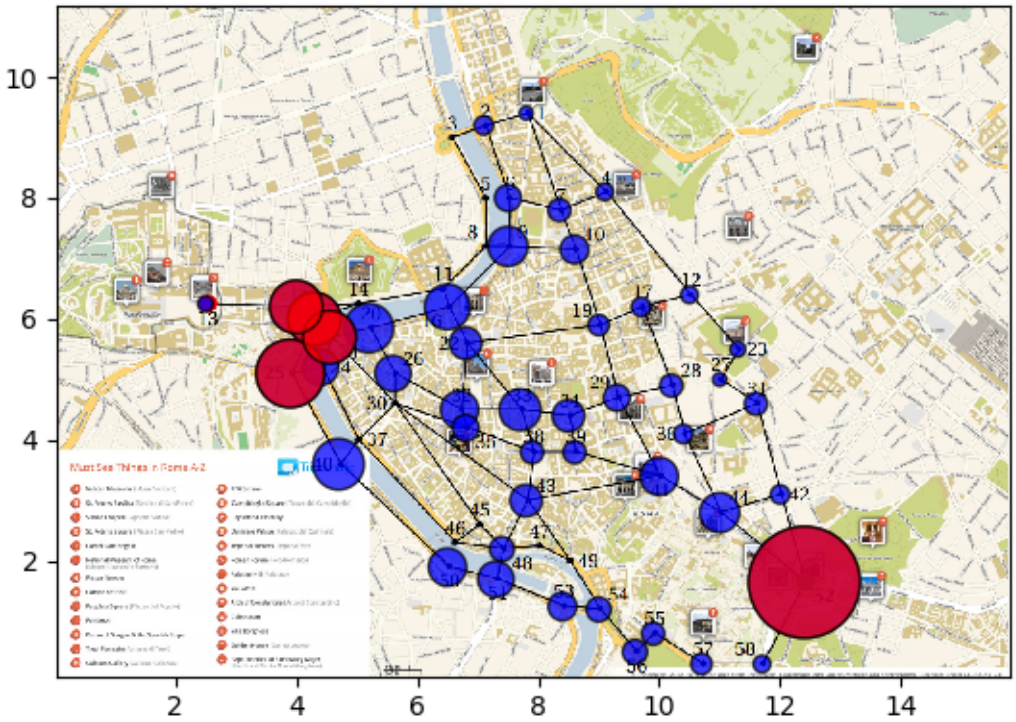


Figure 8.1: Map of maxima, node 30 inaccessible.

Figure 8.1 reveals some interesting features of the system. The busiest nodes in our original scenario became strangley less congested when removing node 30. However the maxima of some of the less used nodes grew. The exact difference between the two cases is printed below. The fist entry in each list is the node number, and the second is the difference in maxima between the two scenarios:
Nodes that increase in congestion:
[22 5] [ 2 5] [10 4]
[ 1 3] [17 2] [28 1]
[19 1] [33 1] [12 1]
[27 1] [23 1]
Nodes that decreased in congestion:
[20 13] [45 13] [43 13]
[35 12] [24 11] [21 10] [41 9]
[25 8] [39 6] [44 6] [40 6]
[38 5] [50 5] [16 5] [53 4]
[54 4] [37 3] [32 3] [51 3]
[26 3] [29 2] [58 2] [ 6 2]

[ 9 1] [52 1] [48 1] [42 1]
[56 1] [34 1] [55 1] [57 1]
As one can see the number of nodes that decrease in congestion far outweigh the number that increase, and the amount they decrease by is particularly dramatic.
To delve further into this phenomenon, I checked the total number of cars in our system, using the command 'print sum(c)' at each timestep. In the unaltered system, this quantity hovers around 350, when reaching a stable state. Whereas when removing node 30, it is closer to 370.
Both of these points imply that although there are more cars in the system, they are more spread out across it.
In conclusion, we find that removing a particular busy node can decrease congestion across the system by forcing cars to spread out, and use longer routes. The code for the outputs above is wirtten below:

```
        cmax0, used0=flow()
        cmax1, used1=flow(TakeOut=30)
        #Find maximums for the normal system and
        #system without node 30
        diff= cmax1- cmax0
        #calculate the difference
        diff_index = np.argsort(diff)
        #sort the indices in terms of the maximum
        increase, decrease = [], []
        for i in diff_index:
                if diff[i]>0:
                        increase.append([i+1,diff[i]])
                        #Create list of increased maxima
                elif diff[i]<0:
                        decrease.append([i+1, -diff[i]])
                        #List of decreased maxima
        increase=increase[::-1]
        print np.array(increase)
        print np.array(decrease[1:])
        #print lists, removing node 30
        plot(cmax1)
        #plot the system without node 30
```

## 9 Conclusion and Improvements

In conclusion I have successfully created a model that simulates the traffic flow through a city during rush hour. I have stumbled across and discussed some interesting points of the model, and gained insight into traffic network modelling in general. In retrospect, one of the biggest features I would change in my model, is how it uses Dijkstra's Alogrithm for

every node to find the optimum paths. This is not necessary, and can be slow for a large number of nodes.
A quicker possible method would be to run dijkstra from the destination, to the beginning point, then reversing the outputted path. In theory I would only have to run Dijkstra's Algorithm once. This is becuase a key feature of the algorithm, is that it must calculate the shortest path to all nodes, in order to find a shortest path to a specific node. By changing the nature of the output of the algorithm, I could create a spanning tree starting at the end node, giving the optimal paths without looping over every vertx. This would therefore be a lot quicker. I will try and implement this in the future.