IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

M3SC Project 1: Traffic Flow Through A Simplified Street Network

Christopher McLeod

February 22, 2017

Abstract

This project implements a simple simulation of traffic flow on a network. We imagine modeling the movement of cars through this network where each car is making choices according to the local optimal route as determined by its GPS-navigation system. An attempt was also made to make the approach slightly more general, done by extending the model to any network. For this I use the concept of an injection node and exit node, and generalise the code to work for any network by allowing csv files to be variable. From this, a general discussion is made about the results of the simulation.

1 Constructing the Main Code

Four functions are responsible for the simulation of movement of the cars through the network. 3 of these are in supplement to the main program: *RomeFlow.py*. The division of these functions into to two modules is made explicit in fig. 1 below

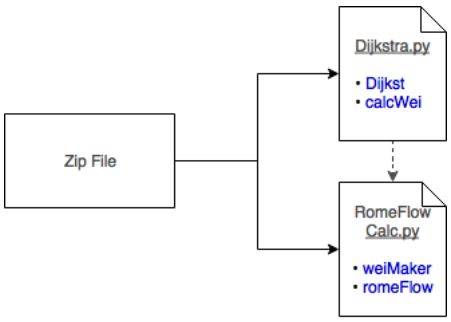

Figure 1.1: File Flow

Dijkst is the exact code as written by Prof. P. Schmid, as is calcWei, bar renaming of variables. For this reason, these two functions are omitted here, but are given for completion in the appendix. Dijkstra finds the least path via Dijkstra's algorithm and calcWei calculates the weight matrix. weiMaker is code lifted from Prof. P. Schmid, that has been abstracted into a function that takes two CSVs, giving the position of the vertices and the weight of the edges. It interprets them and returns, using calcWei, the weight matrix for the network described by the two files.

```
1  # create a function to create a weight matrix from two cvs.
2  # cvs expected to give the position of the vertices (vertPos)
3  # and edge weights(edgeWei) of the network
4  def weiMaker(vertPos, edgeWei):
5
6      X = np.empty(0,dtype=float)
7      Y = np.empty(0,dtype=float)
8      with open(vertPos,'r') as file:
9          AAA = csv.reader(file)
10         for row in AAA:
11             X = np.concatenate((X,[float(row[1])]))
12             Y = np.concatenate((Y,[float(row[2])]))
13     file.close()
14
15     A = np.empty(0,dtype=int)
16     B = np.empty(0,dtype=int)
17     V = np.empty(0,dtype=float)
18     with open(edgeWei,'r') as file:
19         AAA = csv.reader(file)
20         for row in AAA:
```

```
            A = np.concatenate((A,[int(row[0])]))
            B = np.concatenate((B,[int(row[1])]))
            V = np.concatenate((V,[float(row[2])]))
    file.close()

    wei = Dijkstra.calcWei(X,Y,A,B,V)

    return wei
```

Above is the code in weiMaker responsible for returning the weight matrix. The point of this function is that it is general enough to work for any CSV files describing a network, so long as they are in similar form to the ones we use. As mentioned, this is code is courtesy of Prof. P. Schmid and is not explained here.
Next we look at the rest of the RomeFlow.py file, starting with the package list

```
# -*- coding: utf-8 -*-
import numpy as np
import csv
import copy as cp
import Dijkstra
```

```
# main flow simulating code
# takes csvs to make initial weight matrix, has additional
# parameter xi
# the coefficient in updating the weight matrix, and ex,
#the exit node (51 here)
def romeFlow(vertPos, edgeWei, xi, inj , ext ):

    # initialise params; init. weight matrix and make a copy
    # to be updated
    w = weiMaker(vertPos, edgeWei)
    wei = cp.copy(w)
    C = np.zeros((58,201))
    moving = np.zeros(58)

    # initialise the system, before any iterations
    C[inj,0] = 20

    # begin the time loop of 200 iterations
    for t in range(200):
        # predetermine the amount of cars to move
        for i in range(len(moving)):
            moving[i] = round(0.7*C[i,t])

        # loop through nodes and move them to next best
```

```
        for n in range(58):
            if (C[n,t] > 0 and n != ext):
             # find next move: next node on shortest path
             # to exit node
                shpath = Dijkstra.Dijkst(n, ext, wei)
                nm = shpath[1]
             # update n by removing from it the cars that
             # are moving
                C[n,t+1] += (C[n,t] - moving[n])
             # update the next move node with the cars
             # moving from n
                C[nm,t+1] += moving[n]
            elif n== ext :
                # let 40% of the cars exit
                C[n,t+1] += np.round(0.6*C[n,t])



        # inject 20 cars into the injection node for the
        # first 180 iterations
        if t<179 :
            C[inj,t+1] += 20

        # update the weight matrix co-ordinate wise, making
        #sure node i and j
        # are truly connected, first
        for i in range(58):
            for j in range(58):
                if w[i,j] > 0 :
                    wei[i,j] = w[i,j] + xi*0.5*(C[i,t+1]
                               + C[j, t+1])

    # return C, the 'flow matrix'
    return C
```

I give now a brief explanation of the code. romeFlow ties together all the previous functions in one, it takes as its arguments the position of the vertices, weight of the edges, xi, the injection node and exit node. The aim of the function is to then return a $58x201$ matrix which is the number of cars at any node at any time point, i.e. $C_{n,t} = \#$ of cars at node n at time t. The 58 columns represent the nodes from 1 to 58 and the rows represent the time steps. The at $t = 0$ we have the initial system with 20 cars at node the injection node (pythonic index). From here 200 iterations are carried out in time steps, subject to the model.

In each time step, the number of cars to move in the next time step is calculated. The nodes are then looped through. if a car is at a node, $n \neq ext$, we moving the pre-calculated

number of cars to the next node on the (currently) shortest path to the exit node. The node to which it is moving is updated for the next time step with its future value add the number of cars moving from n; at n we update the value at the next time step with its future value add the current value minus those moving from n. If, however, we're at ext we update the future value by itself plus 60% of the current value, simulating an exit rate of 40%.

The code also checks that 180 iterations have not already been carried out, and if so it injects 20 cars into the system one time-step in the future, ready to be moved in the next iteration. Lastly, the weight matrix is updated as according to the formula, ready for use in the next iteration.

Now the basic code has been formed, the results are ready to be analysed. This code forms a framework which will be exploited and perturbed in the next section of this paper.

2 Analysis of the Model

In this section, the results shall be analysed for the Rome network, for 20 cars entering node 13 and flowing through the network to exit node 52. From here we seek to determine for each node the maximum load; identify the most congested nodes; see which edges are not utilised at all and why. Next we analyse the flow pattern for parameter $xi = 0$. Lastly an accident at node 30 is introduced and the flow is again considered under this perturbation.

2.1 Maximum Load

We seek the maximum load at each node for all time. Over the 200 iterations, taking the maximum of the returned matrix in the first axis (time) gives a vector of 58 values which details the largest number of cars at this point in time. Effecting this in code, RomeFlow-Calc.py is appended with

```
1 if __name__ == '__main__':
2     print 'array of max number of cars at node n=1->58:',
3           np.max(romeFlow('RomeVertices', 'RomeEdges(2)', 0.01,
4                  12,51), axis = 1)
```

Which returns

```
array of max number of cars at node n=1->58:
[  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.]
```

2.2 Congestion

The most congested nodes can be identified in a similar manner. The most congested nodes are the ones for which over all time, the number of cars which passed through it

are highest. To do this, sum across the first axis of the return matrix. Practically written by appending the RomeFlowCalc py file with

```
if __name__ == '__main__':
    cong = np.sum(romeFlow('RomeVertices',
          'RomeEdges(2)', 0.01,12,51), axis=1)
    print 'greatest congestion occurs at nodes:',
                  cong.argsort()[-5:][::-1]
```

cong is the sum across the nodes across all time and argsort orders the indices of cong from smallest to largest in terms of the magnitude of the value at that index. This yields

```
greatest congestion occurs at nodes: [51 17 12 14 20]
```

the $[-5:]$ gives the indices of the last 5 elements of the sorted array, giving the 5 most congested nodes.

2.3 Edge Utilisation

We seek the edges in the network which have at no time been traversed. To do this requires a small alteration to the main code in romeFlow. The idea here is to return a 'use-matrix' which has value 1 at co-ordinate (i, j) for which $w_{i,j} \neq 0$ (the weight matrix) and $i \not\rightarrow j$ (i.e no car goes from node i to node j) and zero everywhere else. From this we can simply sum the number of '1's.

Begin by adding, to the top of the function, code which initialises Uu, the use-matrix of ones where there is a one in w

```
    # create Uu to count which edges get used
    Uu = cp.copy(w)

    # set the non-zero values in the weight matrix to 0 in Uu
    for y,i in np.ndenumerate(Uu):
         if i>0:
             Uu[y] =1
```

Here a copy is made of the initial weight matrix. The numpy function ndenumerate returns the co-ordinate pair $y = (i, j)$ and the value at y, i; This is then used to set the non-zero values to 1.

Next, in the node loop, because we move cars from node n to $nm = dijkst(n, 51, wei)$ we set $(Uu)_{n,nm} = 0$ as $n \rightarrow nm$. This is implemented like so

```
                shpath = Dijkstra.Dijkst(n, ext, wei)
                nm = shpath[1]
                # set used edges to zero
                Uu[n,nm] = 0
```

The final amendment is to the last line of the code, to make sure that the boolean use-matrix is returned also

```
    # return C, the 'flow matrix'
    # return matrix of unused edges
    return C, Uu
```

Now, the code to sum the use-matrix is simply appended, as is a quick list comprehension which uses ndenumerate to give a list of (i, j) such that $i \not\rightarrow j$

```
if __name__ == '__main__':
    _,Uu = romeFlow('RomeVertices', 'RomeEdges(2)', 0.01,12,51)
    print 'number of unused edges:', np.sum(Uu)
    print 'on edges:', [y for y, i in np.ndenumerate(Uu)
                                                if i == 1]
```

This give the result

```
number of unused edges: 71.0
on edges: [(0, 1), (0, 6), (1, 2), (2, 1), (2, 4), (3, 0), (3, 6),
(4, 7), (6, 0), (6, 5),(7, 8), (7, 10), (8, 7), (9, 8), (10, 13),
(10, 15), (11, 3), (13, 14), (13, 17), (14, 12),(15, 10), (16, 18),
(17, 14), (18, 9), (20, 17), (22, 11), (26, 22), (27, 16), (27, 28),
(28, 18), (28, 33), (29, 20), (30, 26), (30, 35), (31, 21), (31, 25),
(32, 21), (32, 31), (33, 32), (35, 27), (36, 23), (37, 34), (38, 37),
(40, 28), (40, 38), (40, 42), (41, 30), (41, 43), (42, 29), (42, 47),
(42, 48), (43, 35), (43, 40), (43, 41), (44, 29), (44, 45), (45, 36),
(45, 44), (46, 47), (47, 44), (47, 45), (48, 42), (48, 46), (51, 41),
(51, 43), (51, 57), (53, 48), (54, 55), (55, 53), (56, 54), (57, 56)]
```

In light of this result, it is worth noting here that $i \rightarrow j \not\Rightarrow j \rightarrow i$. For example, from node 13 we may send cars to node 15, but Dijkstra will surely never send cars back to 13 from 15. So to say 71 edges are unused is an overstatement if we do not consider the direction of edges separately. If we say an edge is 'utilised' if we send cars $i \rightarrow j$ *or* $jtoi$, then we implement the following code before the end of the function (out of any loops)

```
    # count edges as utilised if they are not one way and
    # have been used in either direction
    # delete double entries
    for i in range(58):
        for j in range(58):
            if (w[i,j] == w[j,i] and Uu[i,j] != Uu[j,i]):
                Uu[i,j] = 0
            if U[i,j] == U[j,i]:
                Uu[i,j] == 0
```

Note, the condition $w_{i,j} == w_{j,i}$ checks the edge is not one-way, while the condition $U_{i,j} \neq U_{j,i}$ highlights the edges which have been traversed in one direction. The result is now given

```
number of unused edges: 13.0
on edges: [(2, 4), (4, 7), (7, 10), (10, 13), (13, 14), (13, 17),
(20, 17), (36, 23), (45, 36), (46, 47), (47, 45), (48, 46), (53, 48)]
```

This is the concise list of unused edges. Shown highlighted in the image below

The reason these edges were not used is due to them not being viable next moves in Dijkstra's algorithm. Either they were too far away or had too high a cost to travel them, even given the perturbations on the weight matrix. For example, the weight from

2.4 THE SYSTEM FOR $\xi = 0$