IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

M3SC Project 1

Changmin Yu

February 20, 2017

1 PREPARATIONS

1.1 IMPORT MODULES

We first need to import a few modules we will be using in the project:

```
1 import sys
2 import numpy as np
3 import math as ma
4 import csv
```

1.2 DIJKSTRA'S ALGORITHM

We need the Dijkstra's Algorithm provided in the lectures to compute the optimal path from one node to another based on the weight matrix:

```
1 def Dijkst(ist,isp,wei):
2     # Dijkstra algorithm for shortest path in a graph
3     #     ist: index of starting node
4     #     isp: index of stopping node
5     #     wei: weight matrix
6 
7     # exception handling (start = stop)
8     if (ist == isp):
9         shpath = [ist]
```

```
10          return shpath
11
12      # initialization
13      N         =  len(wei)
14      Inf       =  sys.maxint
15      UnVisited =  np.ones(N,int)
16      cost      =  np.ones(N)*1.e6
17      par       = -np.ones(N,int)*Inf
18
19      # set the source point and get its (unvisited) neighbors
20      jj            = ist
21      cost[jj]      = 0
22      UnVisited[jj] = 0
23      tmp           = UnVisited*wei[jj,:]
24      ineigh        = np.array(tmp.nonzero()).flatten()
25      L             = np.array(UnVisited.nonzero()).flatten().size
26
27      # start Dijkstra algorithm
28      while (L != 0):
29          # step 1: update cost of unvisited neighbors,
30          #         compare and (maybe) update
31          for k in ineigh:
32              newcost = cost[jj] + wei[jj,k]
33              if ( newcost < cost[k] ):
34                  cost[k] = newcost
35                  par[k]  = jj
36
37          # step 2: determine minimum-cost point among UnVisited
38          #         vertices and make this point the new point
39          icnsdr     = np.array(UnVisited.nonzero()).flatten()
40          cmin,icmin = cost[icnsdr].min(0),cost[icnsdr].argmin(0)
41          jj         = icnsdr[icmin]
42
43          # step 3: update "visited"-status and determine
44          # neighbors of new point
45          UnVisited[jj] = 0
46          tmp           = UnVisited*wei[jj,:]
47          ineigh        = np.array(tmp.nonzero()).flatten()
48          L             = np.array(UnVisited.nonzero()).flatten().size
49
50      # determine the shortest path
51      shpath = [isp]
52      while par[isp] != ist:
53          shpath.append(par[isp])
```

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

    return shpath[::-1]
```

1.3 Calculating Weight Matrix

Now we need a function to calculate the weight matrix:

```
def calcWei(RX,RY,RA,RB,RV):
    # calculate the weight matrix between the points

    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
```

Then, we use the following code to extract data from the two files provided and then calculate the initial weight matrix in the main function:

```
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()
# extract the data from our files and generate RX, RY, RA, RB, RV

wei = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV) # initial weights
```

1.4 Update the weight matrix

I will use the following function to update the weights of all segments w_{ij}, i.e, update the weight matrix for the network, at each of the 200 iterations:

```
def updateWei(wei,c,xi):
    # this function is for updating the weights of all
    # segments w_{ij} in the network given ta initial weight
    # matrix, an array of number of cars at all 58 nodes and
    # a parameter xi
    m, n = np.shape(wei)
    upwei = wei.copy()
    for i in range(m):
        for j in range(n):
            if (upwei[i, j] != 0):
                # if currently there is no weight between
                # node i and j, we cannot update the weight
                # w_{ij} since we cannot go straight from i
                # to j
                upwei[i, j] = wei[i,j]+xi*(c[i]+c[j])/2
                # update the weights using the given
                # function upwei[i, j] += ip*(c[i]+c[j])/2
    return upwei
```

This function is easy according to the update rules given in the project, one thing worth noticing is that we only update the weights between i and j if there were already an edge between them in the original matrix, we cannot update two nodes which are not connected despite that there are cars currently at these two nodes.

1.5 Rome Network function

I will be using the following function to compute the maximum car load for each node over the 200 iterations, select the top five most congested nodes based on the total car loaded of each node, and the edges not utilized during the 200 iterations. This function is the most important function we are going to use in this project (in which I add extensive comments to make my function as clear as possible):

```
def Rome_Network(wei, xi):
    # we are going to use this function to calculate the
```

```
    # maximum loads at each node in the network over the 200
    # iterations, the five most congested nodes and the edges
    # not utilized
    n = len(wei)
    current_load = np.zeros(n, int)   ####################
    max_load = np.zeros(n, int)       #  initialization  #
    Wei = wei.copy()                  ####################
    U = np.zeros((n, n), int)
    # initialise a zero matrix, which will play a role of an
    # indicator matrix, i.e, if U[i, j] == 0, it means that
    # there are no cars move from i straight to j over the 200
    # iterations, and by comparing with the initial weight
    # matrix we can have a list of edges not utilized
    for i in range(1,201):
        move = np.round(current_load*0.7)
        move[51] = np.round(current_load[51]*0.4)
        move = move.astype(int)
        # generate the list of number of cars move to the next
        # code of each nodes in the network, special attention
        # paid to the exit node 52 (since only 40 percent
        # leave at this node), and make the list a list of
        # integers
        arrive = np.zeros(n, int)
        # initialising the number of cars arrive at each of
        # the 58 nodes

        for j in range(n):
            if (current_load[j] != 0):
                path = Dijkst(j, 51, Wei)
                if (len(path) != 1):
                    arrive[path[1]] += move[j]
                    U[j, path[1]] += 1
                else:
                    pass
        # this for loop updates the list "arrive" and the
        # matrix U after using the Dijkstra's algorithm to
        # compute the optimal path to node 52 from each node
        # of the network except for node 52

        current_load += arrive-move
        # update the car load at each of the nodes

        if i <= 180:
            current_load[12] += 20
```

```
        else:
            pass
        # this if-statement ensures that 20 cars are injected
        # at the entrance node 13 for the first 180 minutes

        for k in range(n):
            if max_load[k] < current_load[k]:
                max_load[k] = current_load[k].copy()
            else:
                pass
        # this for loop updates the maximum number of cars
        # each node loads after each iteration

        Wei = updateWei(wei, current_load, xi)
        # use the updateWei function to update the weights of
        # the network after updating the car-load of each
        # nodes in the network

    most_cong = max_load.argsort()[::-1][:5]
    # use argsort() to obtain the indices of the top five
    # nodes based on the number of cars total loaded by each
    # node

    n_utilized = []
    # initialisation of a list of edges not utilized during
    # the iterations
    for i in range(n):
        for j in range(n):
            if U[i, j] == 0 and wei[i, j] != 0:
                n_utilized.append([i+1, j+1])
    # the for loop is selecting indices i and j such that no
    # cars move straight from i to j and there is indeed a
    # path between i and j, then add the path [i, j] to the
    # list "n_utilized"

    return max_load, most_cong, n_utilized
```

The general strategy of the Rome_Network function is that for each iteration over the 200 iterations, we update the number of cars leave node i and the number of cars arriving at node i for all i, then after using np.round() to round our results, if we are not starting at node 52, we update the arrive list using the move list according to the moving rules of the cars, then update the maximum load of each node by comparing the new and current car-load at each node. Finally, we use the updateWei function stated above to update the weight matrix of the network according to the rule given in the project. At each iteration, we use Dijkstra

to compute the optimal route from each node to node 52 (python-index: 51), and we look for the 1-entry of the route, if exists, we add one to the [i, path(i)[1]] entry to the matrix U, which means that the edge between node i and node path(i)[1] is utilized, and after all iterations, the zero entries [i, j] indicate that no cars travel straight from node i to node j, then by comparing with the original weight matrix, which gives an indication for whether there is an edge between node i and node j, we can obtain a list of edges not utilized over the iterations. Finally, after all200 iterations, we obtain the final max_load array, then we use arsort() function to sort the list to obtain the five most congested nodes over the 200 iterations.

2 Report

2.1 Question 1 & 2 & 3

For the first three questions of the project, I will use my Rome-Network function, given the original weight and the parameter $\xi = 0.01$:

```
1 max_load_ori,most_cong_ori,n_utilized_ori=Rome_Network(wei,0.01)
```

2.1.1 Maximum Load

The maximum load of each node over the 200 iterations is:

[4 6 0 7 0 10 8 0 15 13 0 9 28 0 28 20 7 28 10 27 37 11 7 23 39 22 6 11 13 34 13 16 14 14 19 10 0 14 14 29 30 12 32 27 0 0 0 11 0 22 17 59 15 14 12 13 11 11]
where the i^{th} element in the list corresponds to the maximum load of the i^{th} node over the 200 iterations.

2.1.2 Most Congested nodes

The five most congested nodes based on the total number of cars loaded at each node are:

[52 25 21 30 43]

2.1.3 Edges not utilized

The edges not utilized over the 200 iterations are:
[[1, 2], [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], [18, 15], [19, 10], [21, 18], [23, 12], [27, 23], [28, 17], [28, 29], [29, 19], [29, 34], [30, 21], [30, 45], [31, 27], [31, 36], [32, 22], [32, 26], [33, 22], [33, 32], [34, 33], [35, 30], [36, 28], [37, 24], [37, 40], [38, 35], [39, 38], [40, 37], [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], [45, 48], [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]]

where each element of the list is a pair with the form of [i, j], which means that the edge between node i and node j is not utilized at all.
The reason that these edges are not utilized at all is that the weight of these edges in the original weight matrix is very large, even with updating the weights, due to the small value of the parameter ξ, we still choose to omit the edges with large weights throughout the 200 iterations. So for any element in the list above, e.g, [i, j], when Dijkstra chooses the optimal route from i to node 52, it will omit choosing the path i-j due to the large weight of edge ij.

2.2 The parameter $\xi = 0$

We first need to compute the maximum load, most congested nodes and edges not utilized when $\xi = 0$ like the previous case:

```
max_load_new,most_cong_new,n_utilized_new=Rome_Network(wei,0.)
```

Note that in this case, the parameter ξ is zero, hence, by the given rule for updating the weights:

$$w_{ij} = w^0_{ij} + \xi \frac{c_i + c_j}{2}$$

we know that the weight matrix of the network will remain unchanged over the 200 iterations, i.e, we will now deal with the problem whilst always using the original weight matrix computed in Part(1.3) above.
We now notice that from the results of this case, the list of maximum load of each node has many more zero elements and the non-zero elements occur at nodes: 13, 15, 18, 25, 40, 50, 51, 52, 53, 54, 55, 56, 57, 58. Now, we use Dijkstra to compute the optimal route from node 13 to node 52 (python indices 12 and 51):

```
Dijkst(12, 51, wei)
```

and the optimal route is:
[12, 14, 17, 24, 39, 49, 50, 52, 53, 55, 54, 56, 57, 51] in python-indices, which corresponds to the nodes with non-zero maximum car loads in the list "max_load_new".
Hence, I conclude that, when the parameter $\xi = 0$, we will then be using the original weight matrix for each iteration, and every car follow the same route/flow pattern during each iteration, and the flow pattern is:

13 -> 15 -> 18 ->25 -> 40 -> 50 -> 51 -> 53 -> 54 -> 56 -> 55 -> 57 -> 58 -> 52

2.3 After the Accident

In this case, an accident occurs at node 30 (python-index 29), and every route from or to node 30 is blocked, so I will alter the original weight matrix by transforming the 30-th row and 30-th column to a list of zeros, and I will doing this using the following code:

```
block_wei = wei.copy()
block_wei[29,:] = np.zeros(58)
```

```
block_wei[:,29] = np.zeros(58)
# update the weight matrix when the accident happens and
# all routes to or from node 30 are blocked
```

The block_wei matrix is our new initial weight matrix in this case.
Now we wish to find the most congested nodes and there maximum loads, nodes decrease the most in peak value and nodes increase the most in peak value after the accident occurs, I will be using the following code to compute the above target results:

```
max_load_acc,most_cong_acc,_=Rome_Network(block_wei,0.01)
max_load_most_cong_acc = max_load_acc[most_cong_acc]
# compute the maximum load of the most congested nodes
diff = max_load_acc - max_load_ori
# compute the list of changes in the maximum number of cars
# loaded of each node
dec_most = diff.argsort()[1]+1
# compute the list of nodes decrease the most in peak value
# after the accident, after initial computation, I found
# that the first element of this list is node 30, but we aim
# to exclude node 30 from the list, hence we take the
# second elements after sorting in descending order then
# the [1:6] of the list is our target list
inc_most = diff.argsort()[::-1][0]+1
# compute the node increase the most in peak value
# after the accident
```

Note that I computed the lists of top fives of each targets, if the top-one is wanted, we can just take the python-index 0-entry of each list.
I will use the following code to report the final results of all questions in the project:

```
print 'The maximum load for each node over 200 iterations is:
', max_load_ori
print 'The five most congested nodes are: ', most_cong_ori+1
print 'Edges not utilized: ', n_utilized_ori

print 'The maximum load for each node when xi = 0: ',
max_load_new
print 'The five most congested nodes when xi = 0: ',
most_cong_new+1
print 'Edges not utilized when xi = 0: ', n_utilized_new

print 'After accident, the five most congested nodes are: ',
most_cong_acc+1
print 'After accident, the maximum load of the five most
congested nodes is: ', max_load_most_cong_acc
print 'After accident, the nodes decrease the most in peak
```

```
value are: ', dec_most
print 'After accident, the nodes increase the most in peak
value are: ', inc_most
# printing our final results
```

Now, the most congested nodes after the accident occurs are: [52 21 25 20 13], and their corresponding maximum loads are: [57 39 35 32 28].
Node 43 decrease the most in peak values after the accident.
Node 9 increase the most in peak values after the accident.

————————————————————END OF PROJECT 1 ————————————————————-