IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

# Scientific Computing (M3SC)

Pei Shan Chang
00975081

February 23, 2017

## 1 STRATEGY SOLUTIONS

The main steps for the implementation will be explained in this section, followed by an example in the next section and detailed description with the codes in the *Implementation* section. Important matrices will also been introduced along the steps. Recalling that the *Dijkstra* function and the formula for new weights has been given during lectures, the implementation is shown below.

1. The simulation is carried out for 200 iterations and the number of the cars at each node after each iteration is recorded in a matrix, `NofCars`. Since there are 200 iterations, `NofCars` has 200 columns with 58 rows (each row represents the node). The entry of this matrix represents the number of cars at that iteration on that particular node.

2. For the first 180 iteration, 20 cars are injected at node 13 (`python index 12`). This injection is updated in the vector `CurrentCars`. `CurrentCars` is a *vector* that copies the column of `NofCars` at

`iteration-1`. `CurrentCars` is the flowing vector that changes in every iteration and is used to determine the next column of `NofCars`. Any changes to the number of the cars before entering the calculations for the movement of the cars is recorded in this vector. For example, the 12th entry of vector `CurrentCars` is updated with an additional 20 cars for the first 180 iterations.

3. Next step, the weight at each node is updated. The weights are updated every minute (also means `iteration`) and is stored in a matrix named, `WeiN`. Similar to `wei` matrix, `WeiN` has 58 columns with 58 rows. See *Implementation* section for *Construction of `WeiN`*.

4. This step is the car movements calculation part. At each node that has cars, *Dijkstra* function is applied to get the optimal route to node 52 from the node. Then 70 % of the cars on that node move to the next node while the rest remain. The number of existing cars is taken from the vector `CurrentCars`, and the number of the cars after the movement is recorded as a new entry for `NofCars`.

5. The first iteration for the simulation is initiated manually instead of looping. This is because `CurrentCars` could not take the `iteration-1` column of `NofCars` at the first iteration. Starting from the second iteration (`python index 1`), a *loop* method is used.

6. The loop is done by repeating step (2) to (4) for 199 iterations.

## 2 Example

To give a clearer view for the steps in section *Strategy Solutions*, take iteration = 2. The steps are shown below.

`Iteration =1`

1. 20 cars are injected into the 12th entry of `CurrentCars`, `CurrentCars[12]`.

2. Weights of the nodes with $w_0 \neq 0$ are updated in `WeiN`.

3. *Dijkstra* is then applied to the index in `CurrentCars` that has entries not equal to zero. In this case, it is only used for node 13.
4. The movement of the cars are calculated and recorded in `NofCars` under the first column (`python index 0`). In this case, it means that 12th entry of `NofCars` is 30 % of the 12th entry in `CurrentCars` while 14th entry will have 70 % of it (see line 39-40 below in the *main code*, listing 2).

`Iteration = 2`

1. `CurrentCars` copies the column of `NofCars` at `iteration = 1`.
2. 20 cars are injected in `CurrentCars[12]`.
3. Weights are updated with the same condition.
4. *Dijkstra* is applied to both nodes 13 (`python index 12`) and 15(`python index 14`), one by one. At every node, the movements of cars are recorded in `NofCars` under the second column (`python index 1`).

## 3 Implementation

### 3.1 Weights Construction

Listing 1: WeiN Code Structure

```
#since the weights change at every iteration,
#define a new weight function.
def NewWeight(M):
    #calculate the updated weights.
    for k in range(len(wei)):
        for l in range(len(wei)):
            if wei[k,l]>0: #only update for non-zero weights
                M[k,l]=wei[k,l]+0.5*eta*(CurrentCars[k] \ #formula
                +CurrentCars[l])
    return M
```

Note that `wei` is the original weight matrix calculated using the function `calcWei`. It should be kept constant throughout the simulation. Line 7 gives the restrictions of only update weights on nodes with initial weight, $w_0 \neq 0$. This is because with a zero weight in the matrix means that there is no route connecting the two nodes; neither can the cars pass by there. In addition to that, updating the zero weights will cause weights appearing on the entries which leads to failure of *Dijkstra* calculation.

The main code:

Listing 2: Main Code

```
if __name__ == '__main__':

    import numpy as np
    import scipy as sp
    import csv
#Project 1
    RomeX = np.empty(0,dtype=float) # import data
    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()

    wei = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV) #calculate the weights
    WeiN=np.zeros([len(wei),len(wei)]) #to store new weights
    NofCars=np.zeros([len(wei),200]) #store #ofcars after each iteration
    CarsOut=np.zeros(200) #record #of cars left for verification
    CurrentCars=np.zeros(len(wei)) #copy of NofCars at iteration-1
    eta=0.01 #eta value for weight calculation
    Edges=zip(RomeA,RomeB) #all edges to be removed once it's utilised
    #start iterations
    #for minute=0
    #step 1: inject car
    CurrentCars[12]+=20
    #step 2: update weights
    WeiN=NewWeight(WeiN)
    #step 3: run Dijkstra using new weights
    shpath=Dijkst(12,51,WeiN)
    #step 4: calculate the car movements at the node
    #calculate the outbound car first
    #deduct another one to get the remaining cars
    NofCars[shpath[1],0]=0.7*(CurrentCars[shpath[0]])
    NofCars[shpath[0],0]=CurrentCars[shpath[0]]-\
                         0.7*(CurrentCars[shpath[0]])
   #start the loop
   #cars are injected for first 180 iterations only
   #step 1 to 4 are repeated
    for minute in range(1,200):
   #update the flowing vector.
        CurrentCars=NofCars[:,minute-1].copy() #number of cars at the moment
        if minute in range(1,180): #inject cars
```

```
55              CurrentCars[12]+=20
56          else:
57              pass
58          WeiN=NewWeight(WeiN) #update the weights
59          for car in range(len(NofCars)):
60              if CurrentCars[car]>0: #only run dijkstra for nodes with cars
61                  shpath=Dijkst(car,51,WeiN) #remember to use new weights
62                 #calculate the car movements
63                  if car != 51: #separate working for node 51
64                     NofCars[shpath[1],minute]+=round(0.7* \
65                                      (CurrentCars[shpath[0]]))
66                     NofCars[shpath[0],minute]+=CurrentCars[shpath[0]]-\
67                          round(0.7*(CurrentCars[shpath[0]]))
68                     if (shpath[0]+1,shpath[1]+1) in Edges:
69                        #gather utilised edges & remove them
70                        Edges.remove((shpath[0]+1,shpath[1]+1))
71                  else :  #for node 51 particularly
72                     CarsOut[minute]+=CurrentCars[shpath[0]]-\
73                              round(CurrentCars[shpath[0]]*0.6)
74                     NofCars[shpath[0],minute]+=round(CurrentCars[shpath[0]]\
75                                              *0.6)
76              else:
77                  pass
78      print NofCars #show all the number of cars at every iteration
```

Line 32 to line 47 shows the steps explained in *Example* section. The loop starts at line 48. It follows well the step (2) to (4) explained in *Strategy Solutions*. Line 60 gives the restrictions to apply *Dijkstra* only on nodes with cars at that moment.

Some highlights on the main code:

1. To avoid rounding off errors that would cause inconsistency of the number of cars in the loop, the smaller portion of the cars is calculated such that it is the deduction of bigger portion of the cars

that are supposed to leave or stay, from the total number of cars. This could conserve the total number of cars in the loop (see line 45 to line 47 for example).

2. The function `round` is used to round off the floating numbers to get an integar.

3. At line 53, make sure that `CurrentCars` is fixed as the copy instead of equating them as the later one will lead to inconsistent number of cars for `NofCars[12]`.

4. A operator `"+="` is used to allow add-ons to same entries.

5. The condition at line 63 is necessary in order for the code to run. This is because at node 52, the first node would be 52 while there will be no second node. Therefore, 64 to line 67 would be invalid. Setting up a condition at line 63 can solve the problem.

6. At line 68-70, a new list `Edges` is introduced. This list is used to record the edges that are not utilised. It will be explained at the *Analysis* section.

7. At line 72, a new matrix `CarsOut` is introduced. This matrix is used to record the number of cars that have left the loop during each iteration. `TotalCars` in listing 3 ensure the compatibility with the total number of cars. The expected result for `TotalCars` is a constant increase by 20 at every iteration, until the 180th iteration from which it is constant at 3600 cars.

Listing 3: Verification

```
#check that number of cars is consistent
 CarStayed= np.sum(NofCars,axis=0) #sum by columns
 TotalCars=np.zeros(len(CarStayed)) #store the total #ofcars
 for i in range(len(CarStayed)):
   TotalCars[i]+=CarStayed[i]+np.sum(CarsOut[:i+1])
 print 'The total number of cars in the iteration is',TotalCars
```

To determine the maximum number of car at each node,

Listing 4: maximum number of cars at each node

```
print 'Maximum number of cars at each node is', NofCars.max(axis=1)
```

The code then returns:

```
Maximum number of cars at each node is [ 1.  2.  0.  7.  0.
8.  8.  0.  16.  14.  0.  8.  8.  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.]
```

**Analysis**: Node 13 (`python index 12`) has the maximum number of 8; Node 15(`python index 14`) is 28 and etc. Among all the nodes, the node with the highest maximum number is node 52 with 63 cars.

Further analysing, Listing 5 shows the five most congested nodes. In fact, simply looking at the result above, the top five can be figured out manually.

Listing 5: Five Most Congested Nodes

```
#calculate the most congested nodes
CongNodes=[] #store congested nodes
Max=list(NofCars.max(axis=1)) #make it a list
#sort out the top five
SortedList= sorted(NofCars.max(axis=1), reverse=True)[:5]
for i in SortedList:
   for v,val in enumerate(Max):
      if val==i:  #find the index in max list
         if v+1 not in CongNodes:
            CongNodes.append(v+1)
print 'Five most congested nodes are', CongNodes,
print 'with the corresponding maximum number of', SortedList
```

SortedList is a list that stores the first 5 entries in the sorted Max list. Max list is sorted such that it is arranged from highest maximum number to lowest. Then it is used to find the corresponding index in Max.

**Analysis**: `Five most congested nodes are [52, 25, 21, 30, 43] with the corresponding maximum number of [63.0, 40.0, 38.0, 32.0, 31.0]`

To check which edges are not utilised, a list of edges is created with the name, Edges (line 33 in listing 2). The path from first node to second node is put together and removed from Edges (line 70) once it is determined by *Dijkstra*. Note that the python indexing is always less than one of the actual node number. Therefore at line 68 & 70, 1 is added to the index. The condition stated in line 68 has to hold because some routes are used repeatedly and once it is removed for the first time from Edges, it would not be detected. This leads to invalidity of the code. The result is returned as below.

Listing 6: Edges that are not ulitised

```
print 'There are', len(Edges), 'nodes that are not utilised:', Edges
```

```
There are 72 unutilized nodes:  [(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),
(41, 43), (43, 49), (49, 43), (32, 26), (33, 32), (34, 33),
(29, 34), (38, 35), (39, 38), (41, 39), (41, 43), (43, 30),
(43, 48), (48, 45), (46, 45), (45, 46), (45, 30), (30, 21)]
```

**Analysis**: (i,j) means the edge connecting from node i to node j. The reason why these edges are not utilised could be due to no car visited neither node throughout the simulation and *Dijkstra* is only applied on those have cars. It could be also because of the node that were visited by cars has few options to move to and that edge is just not the optimal one.

What happen when `eta` = 0? The number of cars at the node is not affecting the weights. This leads to the same optimal path being chosen using *Dijkstra* throughout the simulation; that is `12-> 14-> 17-> 24-> 39-> 49-> 50-> 52-> 53-> 55-> 54-> 56-> 57-> 51`. Therefore in the `NofCars` matrix, these nodes are the only nodes containing cars and their maximum number are almost similar, which is with 28 cars except for node 13 (with 8 cars) and node 52 (with 49 cars). The number of edges that are not utilised increases to 143 modes as basically there are only 13 active edges in this case.

Now node 30 (`python index 29`) has been blocked. Hence the weights on this node should be 0 as there will be no car moving in and out the node.

Listing 7: Adjustment made to weights at node 30

```
# Node 30 dead.
wei = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV) #calculate the weights
wei[:,29]=0 #routes connect node 30 =0
wei[29,:]=0
WeiN=np.zeros([len(wei),len(wei)]) #to store new weights
```

Applying the main code after the adjustment on the weights, the following results are obtained.

**Analysis**: `Maximum number of cars at each node is [ 2.  3.  0.
10.  0.  13.  11.  0.  18.  15.  0.  10.  8.  0.  28.  22.
8.  28.  12.  26.  29.  15.  9.  18.  35.  19.  8.  11.  13.
0.  11.  20.  18.  14.  14.  10.  0.  14.  12.  27.  21.  9.
17.  22.  0.  0.  0.  14.  0.  21.  19.  56.  15.  14.  11.
12.  10.  9.]`

The node with highest maximum number of cars is still node 52 (`python index 51`) but the maximum number has dropped to 56 cars.

```
Five most congested nodes are [52, 25, 21, 15, 18] with the
corresponding maximum number of [56.0, 35.0, 29.0, 28.0, 28.0].
```

Notice that the number of cars for all top 5 have dropped as well.

To find the difference between the peak values,

Listing 8: Difference between peak values

```python
#calculate difference
Maxwh=NofCars.max(axis=1) #max of new NofCars
Difference=Maxwh-Max #get the difference between the before and after
print 'Node with the peak value'
print 'increase the most is node', Difference.argmax()+1,
print 'with', Difference.max()
Difference=list(Difference)
Sorted= sorted(Difference, reverse=True)[-2] #get the second one
print 'Node with the peak value'
print 'decrease the most is node', Difference.index(Sorted)+1,
print 'with', -Sorted
```

The following results are obtained.

```
Node with the peak value increase the most is node 6 with 5.0.
Node with the peak value decrease the most is node 43 with
14.0
```