IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC) Project 1

Omar Haque

February 22, 2017

1 MAIN SOLUTION

The code in *solution.py* contains the main program which carries out the process outlined by the question, i.e modelling the process of the cars moving across the city of Rome using the rules described. I have added scripts *to be added* and *to be added* to help answer the related questions at the end of the project.

Below are the imports used by the main program.

```
1 # Imports
2 import numpy as np
3 import csv
4 import sys
5 import math as ma
6
7 # This import is needed for the last question
8 from solution_accident_occurs import max_index_tracker_no30
```

the variable *max_index_tracker_no30* is imported from another python script, *solution_accident_occurs.py* in order to answer one of the questions. This will be discussed in detail later.

Below are the functions required by the program. The docstring's explain their use.

```
# ------------------------------------------------------------------------
# ------------------    FUNCTIONS USED    ------------------------------
# ------------------------------------------------------------------------

def calcWei(RX, RY, RA, RB, RV):
    """
    This function is taken from Tutorials. It calculates the weight matrix
    given information about each node in the system.
    :param RX: The x coordinates of each node in the system
    :param RY: The y coordinates of each node in the system
    :param RA: the connectivity of each node in the system
    :param RB: the connectivity of each node in the system
    :param RV: the speed limits across each edge in the system
    :return: usable weight matrix
    """

    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

def Dijkst(ist, isp, wei):
    """
    This Dijkstra's algorithm implementation is taken from tutorials.

    :param ist: the index of the starting node
    :param isp: the index of the node to reach
    :param wei: the assosciated weight matrix
    :return:
    """

    # exception handling (start = stop)
    if ist == isp:
        shpath = [ist]
        return shpath
```

```
44
45      # initialization
46      N = len(wei)
47      Inf = sys.maxint
48      UnVisited = np.ones(N, int)
49      cost = np.ones(N) * 1.e6
50      par = -np.ones(N, int) * Inf
51
52      # set the source point and get its (unvisited) neighbors
53      jj = ist
54      cost[jj] = 0
55      UnVisited[jj] = 0
56      tmp = UnVisited * wei[jj, :]
57      ineigh = np.array(tmp.nonzero()).flatten()
58      L = np.array(UnVisited.nonzero()).flatten().size
59
60      # start Dijkstra algorithm
61      while (L != 0):
62          # step 1: update cost of unvisited neighbors,
63          #         compare and (maybe) update
64          for k in ineigh:
65              newcost = cost[jj] + wei[jj, k]
66              if (newcost < cost[k]):
67                  cost[k] = newcost
68                  par[k] = jj
69
70          # step 2: determine minimum-cost point among UnVisited
71          #         vertices and make this point the new point
72          icnsdr = np.array(UnVisited.nonzero()).flatten()
73          cmin, icmin = cost[icnsdr].min(0), cost[icnsdr].argmin(0)
74          jj = icnsdr[icmin]
75
76          # step 3: update "visited"-status and determine neighbors of new point
77          UnVisited[jj] = 0
78          tmp = UnVisited * wei[jj, :]
79          ineigh = np.array(tmp.nonzero()).flatten()
80          L = np.array(UnVisited.nonzero()).flatten().size
81
82      # determine the shortest path
83      shpath = [isp]
84      while par[isp] != ist:
85          shpath.append(par[isp])
86          isp = par[isp]
87      shpath.append(ist)
```

```
88
89      return shpath[::-1]
90
91  def next_node(path):
92      """ Returns the next index (after the node itself) in the path.
93          If the path contains only one node, returns the node itself.
94      """
95      if len(path) == 1:
96          return path[0]
97      else:
98          return path[1]
99
100
101 def update_weight_matrix(epsilon, c, original_weight_matrix, noNodes=58):
102     """
103     This function updates the weight matrix according to step 5 of the
104     Project. Note the added fix - the weight matrix is not changed if
105     the original entry was 0.
106
107
108
109     :param epsilon: given in question
110     :param c: the vector containing number of cars at each node
111     :param original_weight_matrix: the weight matrix given by RomeEdges
112     :param noNodes: number of nodes in the system
113     :return: the updated weight matrix
114     """
115     new_weight_matrix = np.zeros((noNodes, noNodes))
116     for i in range(noNodes):
117         for j in range(noNodes):
118             if original_weight_matrix[i, j] != float(0):
119                 new_weight_matrix[i, j] = original_weight_matrix[i, j] + \
120                                           (epsilon * (float(c[i]) +
121                                                       float(c[j]))) / float(2)
122     return new_weight_matrix
123
124
125 def extract_data():
126     """
127     This function opens the RomeVertices and RomeEdges files, and creates
128     global variables RomeX, RomeY, RomeA, RomeB and RomeV. These are variables
129     used to create the original weight matrix.
130
131     """
```

```
    global RomeX, RomeY, RomeA, RomeB, RomeV
    RomeX = np.empty(0, dtype=float)
    RomeY = np.empty(0, dtype=float)
    with open('./data/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('./data/RomeEdges2', '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()
```

Now using these functions, we can execute the main program.

```
    # ------------------------------------------------------------------
    # -------------------    Main program     ------------------------
    # ------------------------------------------------------------------


if __name__ == '__main__':

    # Import the rome edges file
    extract_data()

    # Use the calcWei function from tutorials, along with the data set given
    # to calculate the weight matrix. Also create a copy which is the
    # temporary weight matrix.
    weight_matrix = misc.calcWei(RomeX, RomeY, RomeA, RomeB, RomeV)
    temp_wei = weight_matrix.copy()

    # Initialise minutes and number of nodes
    minutes = 200
    total_nodes = weight_matrix.shape[0]

    # Need a vector carNumbers which stores the number of cars at each vertex
    # in the graph.
    cars_at_node = np.zeros(total_nodes, dtype=int)
```

```
    cars_at_node_updated = cars_at_node.copy()  # cars_at_node updated is simil
    max_cars_at_node = cars_at_node.copy()  # max_cars_at_node is similar

    # To find the edges utilised, we need a 58x58 matrix of
    # False's. We will set each element to True if we move
    # cars from node i to node j.
    edge_utilised = np.zeros((total_nodes, total_nodes), dtype=bool)

    # Iterate through the 200 minutes
    for i in range(minutes):

        # Apply Dijkstra's algorithm to find the fastest path to node 52 in
        # the system. Then use next_node to find the next node in the given
        # path. (step 1)
        next_nodes = [next_node(Dijkst(node, 51, temp_wei))
                      for node in range(total_nodes)]

        # Move all cars as in steps 2,3. Iterate through every node in the
        # system to do this.
        for j_node in range(total_nodes):

            if j_node == 51:
                # We remove 40% of cars from node 52.
                cars_at_node_updated[51] += int(round(cars_at_node[51] * 0.6))
            else:

                # Initialise the number of cars at node j_node.
                number_of_cars = cars_at_node[j_node]

                # Initialise the next node to move to.
                node_to_move_to = next_nodes[j_node]

                # 70% of cars will move. to keep the total conserved,
                # the amount staying is just
                # number_of_cars - amount_moving
                amount_moving = int(round(0.7 * number_of_cars))
                amount_staying = number_of_cars - amount_moving

                # We now update cars_at_node.
                cars_at_node_updated[j_node] += amount_staying
                cars_at_node_updated[node_to_move_to] += amount_moving

                if amount_moving > 0:
                    # Update edges_utilised matrix
```

```
68                      edge_utilised[j_node, node_to_move_to] = True
69
70          # Now all cars have moved where they need to, we set cars_at_node
71          # to this updated vector, and empty the updated vector for the next
72          # iteration.
73          cars_at_node = cars_at_node_updated.copy()
74          cars_at_node_updated = np.zeros(total_nodes, dtype=int)
75
76          # For the first 180 minutes, 20 cars are injected into node 13.
77          if i <= 179:
78              cars_at_node[12] += 20
79
80          # The temporary weight matrix is updated.
81          temp_wei = update_weight_matrix(0.01, cars_at_node, weight_matrix)
82
83          # We have finished an iteration.
84
85        # Now we calculate the maximum number of cars at each node in the system
86          max_cars_at_node = [max(cars_at_node[node], max_cars_at_node[node])
87                              for node in range(total_nodes)]
```

2 Questions