IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC) Project 2

Omar Haque

March 23, 2017

1 SOLUTION STRATEGY

In this coursework, I will use the algorithmic steps provided to design a workflow and job schedule that minimises the duration of the processes while maximising the number of processes that can be executed in parallel.
Rather than describing a strategy in one section and providing the implementation in another, I will be running through each of the steps and describing the functions used in chronological order, finishing on the main code.

1.1 EXTRACT DATA

In order to make this program as robust and as general as possible, I have written a csv file which contains the data from Table 1.1 of the question, i.e the List of jobs, their duration and dependencies. The function *extract_data* uses the csv module to iterate through every row of the datafile, and combine them into a numpy array.

```
1 import csv
2 import numpy as np
3 import sys
4
5
6 def extract_data(file_name):
7     """
8     This function uses the csv module to extract the 'jobs', 'durations'
9     and 'has to be completed before' columns
```

```
10
11      :param file_name: a csv file containing the data
12      :return: a total_nodesx3 np.array containing the 'jobs', 'durations'
13      and 'has to be completed before' columns
14      """
15      # e.g. file_name = './data/jobslist'
16      job = []
17      duration = []
18      completed_before = []
19
20      # open the file
21      with open(file_name, 'r') as file:
22
23          AAA = csv.reader(file)
24          # iterate through each row
25          for i, row in enumerate(AAA):
26              # add the job number
27              job.append(int(row[0]))
28              # add the duration
29              duration.append(int(row[1]))
30              # if there are jobs to be completed before, add them
31              if len(row) > 2:
32                  completed_before.append([int(node) for node in row[2:]])
33
34              else:
35                  # else add an empty list.
36                  completed_before.append([])
37
38      file.close()
39
40      # combine these together
41      data_frame = np.column_stack((job, duration, completed_before))
42
43      return data_frame
```

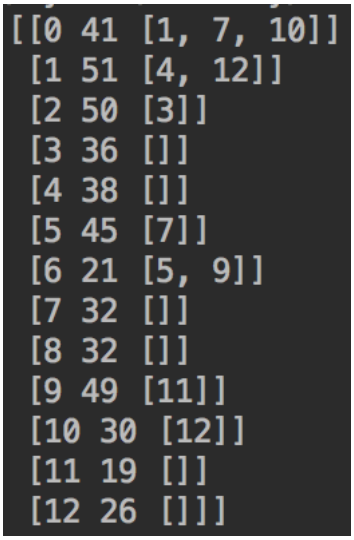
Running this code then produces the array shown in figure 1.1

```
1 data = extract_data('./data/jobslist')
```

1.2 Constructing the graph

We now construct the directed graph, as described in steps **1,2,3** and **4** of algorithm 1.1 in the question. For job $m \in \{0, 1, \ldots, 12\}$ I write m_s and m_f to mean the start and finish node

Figure 1.1: np.array produced from extract_data



for job m respectively. I also write v_s and v_f to mean the virtual start and virtual finish node respectively.
I will construct an adjacency matrix, $W = \{w_{i,j}\}$, where $w_{i,j}$ denotes the edge weight between node i and node j. The ordering of the matrix rows and columns is the following sequence

$$0_s 1_s \dots 12_s 0_f 1_f \dots 12_f v_s v_f$$

The weights $w_{i,j}$ are specified by steps **1,2,3** and **4** of algorithm 1.1, I write the conditions here. $\forall$ jobs m,n $\in \{0, 1, \dots 12\}$:

- w_{m_s,m_f} = the duration of job m. By inspection, the duration of a job is always strictly positive.
- $w_{m_f,n_s} = 0$ if job m has to be completed before job n.
- $w_{v_s,m_s} = 0$
- $w_{m_f,v_f} = 0$

Else, we take a value of $w_{i,j} = -1$ if node i is not connected to node j.
Here is the python implementation:

```
1  def generate_weight_matrix(data):
2      """
3      This function uses the data to create an adjacency matrix, based
4      on the rules outlined.
5
6      :param data: three column np.array with 'jobs', 'durations'
7      and 'has to be completed before' columns
8      :return: the adjacency matrix for this graph
```

```
    """
    global total_nodes, virtual_start, virtual_finish,job_duration
    # we start with an array of -1s and populate the entries that
    # correspond to connected nodes.
    total_nodes = data.shape[0]
    weight_matrix = -1 * np.ones((2 * total_nodes + 2, 2 * total_nodes + 2)
                                 , dtype=int)

    # node start is the index of start jobs
    node_start = data[:, 0].astype(int)
    # job_duration are the job durations for each job
    job_duration = data[:, 1].astype(int)

    # joining each node start to node finish, with the weight as that job's
    # duration.
    weight_matrix[node_start, node_start + 13] = job_duration

    # note on efficiency: I could perhaps do the following for
    # loop by flattening the data[:,2] column, but I need to
    # create a list of node_start corresponding to the number of
    # jobs that each job depends on. This is O(N) anyway, so doing
    # this via a for loop isn't slower.

    # note: also, python doesn't like slicing like a[0,[[4,5],[1,2,3]]]

    for row in data:
        jobs2 = row[2]
        node_start2 = row[0]
        for job in jobs2:
            # this is the connections of dependent jobs
            weight_matrix[node_start2 + 13, job] = 0

    # these are the indices for virtual start and finish
    virtual_start = int(weight_matrix.shape[0]) - 2
    virtual_finish = int(weight_matrix.shape[0]) - 1

    # allow movement between virtual start and all the nodes
    weight_matrix[virtual_start, 0:total_nodes] = 0
    # allow movement from all the nodes to virtual finish
    weight_matrix[total_nodes:virtual_start, virtual_finish] = 0

    return weight_matrix
```

Running this with the data gives us our adjacency matrix, W.

```
weights = generate_weight_matrix(data)
```

1.3 Finding the longest path

We wish to determine the longest path from virtual start to virtual finish. Clearly, the longest path in W is the shortest path in the graph defined by $A := -W$, where $A = \{a_{i,j}\}$. If we look at the definition of W in section 1.2, we see that node i is connected to node $j \iff w_{i,j} \geq 0 \iff a_{i,j} \leq 0$, since $w_{i,j} = -a_{i,j}$. Similarly, node i is not connected to node $j \iff w_{i,j} = -1 \iff a_{i,j} = 1$.
Therefore to find the shortest path in A from v_s to v_f we can apply the Bellman Ford algorithm (since we have negative weights), with the modification that instead of a weight of 0 implying two nodes are not connected, we use 1 instead. I outline these changes below:

Figure 1.2: changes to the Bellman Ford algorithm

```
# step 2: iterative relaxation
for i in range(0, V - 1):
    for u in range(0, V):
        for v in range(0, V):
            w = wei[u, v]
            if (w != 1):
                if d[u] + w < d[v]:
                    d[v] = d[u] + w
                    p[v] = u

# step 3: check for negative-weight cycles
for u in range(0, V):
    for v in range(0, V):
        w = wei[u, v]
        if (w != 1):
            if (d[u] + w < d[v]):
                # print('graph contains a negative-weight cycle')
                pass
```

Otherwise, the code is the same as from tutorials, so I will not include it in full here.

The following code calculates the longest path from v_s to v_f.

```
    adjusted_weights = -1 * weights
    longest_path = updated_bellman_ford(
        virtual_start, virtual_finish, adjusted_weights)[1:-1:2]
```

printing this longest path gives:

Figure 1.3: longest path from virtual start to virtual finish

```
[0, 1, 4]
```

 since our adjacency matrix uses the ordering $0_s 1_s \dots 12_s 0_f 1_f \dots 12_f v_s v_f$, our output will follow the same style. i.e $[v_s, node1_s, node1_f, node2_s, node2_f, \dots, nodek_s, nodek_f, v_f]$. But the virtual nodes are just dummy nodes. We're really trying to find the longest dependent job sequence, and they ensure that we have. So we need to bring our results back to the normal scale, by removing the first and last values, then taking every second value. The python slice [1:-1:2] does exactly this.

So $0 \to 1 \to 4$ is the longest string of jobs.

1.4 Finding the earliest start and stop times

In this section, we adopt the following notation. For a job $m \in \{0, 1, \dots, 12\}$, we write b_m for the path

$$b_m = m_s \to m_f$$

The first thing to realise is that the earliest start time for job $m \in \{0, \dots, 12\}$ is determined precisely by the length of the longest job sequence to m.

This is clear. If job m has no dependencies, it can start right away.
But if there are job sequences that finish on job m, we need to wait for all of these job sequences to finish, before we can execute job m. More formally, if

$$b_{m_1} \to b_{m_2} \to \dots \to b_{m_l} \to b_m$$

is the longest path in the graph ending on b_m , then the earliest time job m can start is $\sum_{i=1}^{l} len(b_{m_i}) = \sum_{i=1}^{l}$ duration of job m_i

Claim: Let the longest job sequence that finishes on job m be $b_{m_1} \to \dots \to b_{m_{l-1}} \to b_m$. Then $\forall i \in \{1, \dots, l-1\}$

the longest job sequence that finishes on job m_i is $b_{m_1} \to \dots \to b_{m_{i-1}} \to b_{m_i}$

Proof: Suppose not. Then there exists a longer sequence to job i: $b_{k_1} \to \dots \to b_{k_p} \to b_{m_i}$, say. But then, $b_{k_1} \to \dots \to b_{k_p} \to b_{m_i} \to b_{m_{i+1}} \dots \to b_{m_{l-1}} \to b_m$ is a longer path to job m than the one we assumed to be. This is a contradiction. ■

This argument about there not existing a longer sequence to job i (or b_i), is the key to this section. And it is why the introduction of the virtual start and finish nodes is so useful, as it means the longest path from virtual start to virtual finish trails through the graph to give you the longest job sequence.

We now have all the information we need to find the earliest start times (the earliest stop time is then just the duration of a job + its earliest start time). The algorithm to do this is as follows:

1. Determine the longest path in the graph. By the arguments above, you have the start and stop times for every job in that path.
2. Those jobs are done. So remove their connection to virtual end, so we never finish on them again
3. If there are still jobs whose start time you haven't determined, GOTO 1. Else, finish.

 We can remove more edges than those specified in 2. But there are no marks for efficiency, so I will leave the algorithm as it is to make it cleaner since we have a very small problem size.
Here is the python implementation for this section.

```
def iterative_bell(adjusted_weights):

    # create a copy of the weight matrix
    temp_weights = np.copy(adjusted_weights)

    # This is a list of 13 Falses
    # If a node appears in a job sequence (i.e. we know its start time)
    # it turns to True.
    removed_nodes = np.zeros(13, dtype=bool)
    # a counter so we know when to stop.
    counter = 0

    while counter < 13:  # O(1)
        # find the longest path in the graph from virtual start
        # to virtual finish
        job_sequence = updated_bellman_ford(virtual_start,
                                            virtual_finish,
                                            temp_weights)[1:-1:2]
        # O(13*E) # all we can change is E


        # remove the connections from those jobs to virtual finish
        temp_weights[np.array(job_sequence)+13,virtual_finish] = 1
```

```
        # determine the times using the formula derived
        current_time = 0  # O(1)

        # iterate through the jobs in the job sequence
        for job in job_sequence:  # O(k)

            # only add to start_times if you haven't already
            if not removed_nodes[job]:  # O(1)
                # set it to the current time. i.e the sum of jobs before it
                start_stop[job, 0] = current_time  # O(1)
                start_stop[job, 1] = current_time + job_duration[job]
                # add 1 to the counter
                counter += 1  # O(1)

            # update current_time
            current_time += job_duration[job]  # O(1)

        # so the for loop is O(k) in total

        removed_nodes[job_sequence] = True  # O(1)
```

This is the code to run this function

```
    start_stop = np.zeros((13, 2), dtype=int)

    iterative_bell(adjusted_weights,start_stop)

    full = np.column_stack((data,start_stop))
```

printing start_stop gives me this as the output

Figure 1.4: earliest start and stop times for each job

```
[[  0  41]
 [ 41  92]
 [  0  50]
 [ 50  86]
 [ 92 130]
 [ 21  66]
 [  0  21]
 [ 66  98]
 [  0  32]
 [ 21  70]
 [ 41  71]
 [ 70  89]
 [ 92 118]]
```

1.5 PRODUCING THE GANTT CHART

2 Main Solution Code

The code in *solution_final.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 *solution_epsilon0.py* and *solution_accident_occurs* to help answer the related questions at the end of the project.

Below are the imports used by the main program.

```
# Imports
import numpy as np
import csv
import sys
import math as ma

# This import is needed for the last question
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. note: I will explain the use of the function away_from_52 properly later in the document.

```
# ------------------------------------------------------------------------
# ------------------    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)
```

```
19      m = len(RA)
20      for i in range(m):
21          xa = RX[RA[i] - 1]
22          ya = RY[RA[i] - 1]
23          xb = RX[RB[i] - 1]
24          yb = RY[RB[i] - 1]
25          dd = ma.sqrt((xb - xa) ** 2 + (yb - ya) ** 2)
26          tt = dd / RV[i]
27          wei[RA[i] - 1, RB[i] - 1] = tt
28      return wei
29
30  def Dijkst(ist, isp, wei):
31      """
32      This Dijkstra's algorithm implementation is taken from tutorials.
33
34      :param ist: the index of the starting node
35      :param isp: the index of the node to reach
36      :param wei: the assosciated weight matrix
37      :return:
38      """
39
40      # exception handling (start = stop)
41      if ist == isp:
42          shpath = [ist]
43          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     """
132     global RomeX, RomeY, RomeA, RomeB, RomeV
133     RomeX = np.empty(0, dtype=float)
134     RomeY = np.empty(0, dtype=float)
135     with open('./data/RomeVertices', 'r') as file:
136         AAA = csv.reader(file)
137         for row in AAA:
138             RomeX = np.concatenate((RomeX, [float(row[1])]))
139             RomeY = np.concatenate((RomeY, [float(row[2])]))
140     file.close()
141     RomeA = np.empty(0, dtype=int)
142     RomeB = np.empty(0, dtype=int)
143     RomeV = np.empty(0, dtype=float)
144     with open('./data/RomeEdges2', 'r') as file:
145         AAA = csv.reader(file)
146         for row in AAA:
147             RomeA = np.concatenate((RomeA, [int(row[0])]))
148             RomeB = np.concatenate((RomeB, [int(row[1])]))
149             RomeV = np.concatenate((RomeV, [float(row[2])]))
150     file.close()
```

```
151
152 def away_from_52(edge):
153     """
154     Tells you whether a given edge is pointing completely away from
155     node 52, in both the x and y directions.
156     :param edge: an edge of the form [a,b]
157     :return: boolean whether or not this points to or away from 52
158     """
159
160     # extract data for access to global variables
161     extract_data()
162
163     # the edge is of the form [a,b]
164     a = edge[0]
165     b = edge[1]
166
167     # use RomeX and RomeY to find the coordinates for a,b and node 52.
168     a_coord = [RomeX[a - 1], RomeY[a - 1]]
169     b_coord = [RomeX[b - 1], RomeY[b - 1]]
170     coord52 = [RomeX[51], RomeY[51]]
171
172     # find the change in x/y from a -> b
173     x_change = b_coord[0] - a_coord[0]
174     y_change = b_coord[1] - a_coord[1]
175
176     # find the change in x/y from a -> 52
177     x_changeTo52 = coord52[0] - a_coord[0]
178     y_changeto52 = coord52[1] - a_coord[1]
179
180     # if we're at 52 we're moving away from it
181     if a == 52:
182         return True
183
184     # if both point in same direction, false.
185     if (x_changeTo52 > 0) and (x_change > 0):
186         return False
187     elif (x_changeTo52 < 0) and (x_change < 0):
188         return False
189
190     # if both point in same direction, false.
191     if (y_changeto52 > 0) and (y_change > 0):
192         return False
193     elif (y_changeto52 < 0) and (y_change < 0):
194         return False
```

```
    # all other tests have passed, so must be True.
    return True
```

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
                    edge_utilised[j_node, node_to_move_to] = True

        # Now all cars have moved where they need to, we set cars_at_node
        # to this updated vector, and empty the updated vector for the next
        # iteration.
        cars_at_node = cars_at_node_updated.copy()
        cars_at_node_updated = np.zeros(total_nodes, dtype=int)

        # For the first 180 minutes, 20 cars are injected into node 13.
        if i <= 179:
            cars_at_node[12] += 20

        # The temporary weight matrix is updated.
        temp_wei = update_weight_matrix(0.01, cars_at_node, weight_matrix)

        # 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)]
```

3 QUESTIONS

1. **Determine for each node the maximum load (maximum number of cars) over the 200 iterations.**

   As we have already incorporated the calculation of maximums in the loop, this is easy:

```
1 max_index_tracker = [[node+1, max_cars_at_node[node]]
2                      for node in range(total_nodes)]
```

   Thus the i'th element of this array gives [node i, maximum number of cars of node i over the 200 iterations]. The code to print is below along with the output.

```
1 print('max_index_tracker is')
2 print(max_index_tracker[0:10])
3 print(max_index_tracker[10:20])
4 print(max_index_tracker[20:30])
5 print(max_index_tracker[30:40])
6 print(max_index_tracker[40:50])
7 print(max_index_tracker[50:(len(max_index_tracker)+1)])
```

Figure 3.1: output of maximum loads

2. **Which are the five most congested nodes?**
   Given this array *max_index_tracker*, we can simply sort by the second argument to find the top five most congested nodes

```
1 top_five = sorted(max_index_tracker,
2                   key=lambda node_and_max: -1 * node_and_max[1])[:5]
3 print('the five most congested nodes are')
4 print(top_five)
```

Figure 3.2: top five most congested nodes output

   In other words, the top five most congested nodes (from highest to lowest) is node 52 with 63 cars, node 25 with 40 cars, node 21 with 38 cars, node 30 with 32 cars and

node 43 with 31 cars.

3. **Which edges are not utilized at all? Why?**

   In the main program we defined a 58x58 boolean matrix of False's, where throughout the process if cars moved from index i to index j, we made the [i,j] element of the matrix True.

   Thus, we have a matrix where indices (corresponding to edges) are *True* if they are traversed by some amount of cars (> 0) in the 200 minute process.

   All that is left to do is count the number of False's, making sure that we don't count a False if the edge couldn't be traversed to begin with (in the original weight matrix). This corresponds to an element of the original weight matrix being 0.

```
# create a boolean matrix that corresponds to the condition
# described above
non_utilised_edges_matrix = (weight_matrix != float(0)) \
                                & (np.logical_not(edge_utilised))

non_utilised_edges = [[i+1, j+1] for i in range(total_nodes)
                          for j in range(total_nodes)
                          if non_utilised_edges_matrix[i, j]]
```

   So an element $[l, m]$ belonging to this vector *non_utilised_edges* implies that the edge $l \to m$ is not utilised.

   The code to print this and the output is below.

```
print('the non utilised edges are')
print(non_utilised_edges[0:10])
print(non_utilised_edges[10:20])
print(non_utilised_edges[20:30])
print(non_utilised_edges[30:40])
print(non_utilised_edges[40:50])
print(non_utilised_edges[50:60])
print(non_utilised_edges[60:(len(non_utilised_edges)+1)])
```

Figure 3.3: list of non utilised edges output

Now I answer *why* these edges aren't utilised.

For the vast majority of unused edges, it is intuitively clear that the reason they are included is because they are pointing completely away from the direction towards

node 52. For example, I have highlighted edges $4 \rightarrow 1$, $12 \rightarrow 4$ and $52 \rightarrow 58$ in figure 3.4.

Figure 3.4: City of Rome graph map with drawn edges

These edges in particular are not only in the wrong in the x direction, but wrong in the y direction too. I have created the function *away_from_52* below to return a boolean depending on whether a given edge is pointing completely away from node 52. i.e. in both x and y direction.

```
1  def away_from_52(edge):
2      """
3      Tells you whether a given edge is pointing completely away from
4      node 52, in both the x and y directions.
5      :param edge: an edge of the form [a,b]
6      :return: boolean whether or not this points to or away from 52
7      """
8
9      # extract data for access to global variables
10     extract_data()
11
12     # the edge is of the form [a,b]
13     a = edge[0]
14     b = edge[1]
15
16     # use RomeX and RomeY to find the coordinates for a,b and node 52.
17     a_coord = [RomeX[a - 1], RomeY[a - 1]]
18     b_coord = [RomeX[b - 1], RomeY[b - 1]]
19     coord52 = [RomeX[51], RomeY[51]]
20
21     # find the change in x/y from a -> b
22     x_change = b_coord[0] - a_coord[0]
23     y_change = b_coord[1] - a_coord[1]
24
25     # find the change in x/y from a -> 52
26     x_changeTo52 = coord52[0] - a_coord[0]
27     y_changeto52 = coord52[1] - a_coord[1]
28
29     # if we're at 52 we're moving away from it
30     if a == 52:
31         return True
```

```
32
33      # if both point in same direction, false.
34      if (x_changeTo52 > 0) and (x_change > 0):
35          return False
36      elif (x_changeTo52 < 0) and (x_change < 0):
37          return False
38
39      # if both point in same direction, false.
40      if (y_changeto52 > 0) and (y_change > 0):
41          return False
42      elif (y_changeto52 < 0) and (y_change < 0):
43          return False
44
45      # all other tests have passed, so must be True.
46      return True
```

We then run this to find the number of edges facing completely away from node 52.

```
1  new_unused = list(non_utilised_edges)
2
3  for _, edge in enumerate(non_utilised_edges):
4          if away_from_52(edge):
5          new_unused.remove(edge)
6
7  print('length of new_unused is %i' % len(new_unused))
```

Figure 3.5: length of new_unused list

So out of 71 unused edges, 41 of them were facing completely away from node 52. In a similar way we can remove nodes that are facing away from node 52 in the x direction only.

I print the remaining edges below

```
1      print(new_unused[0:10])
2      print(new_unused[10:20])
3      print(new_unused[20:31])
```

Figure 3.6: list of unused edges that aren't facing completely away from node 52

We note that from the first question where we found the maximum load for all cars over the 200 iterations, it is clear that nodes with maximum load 0 (i.e. node 3, 5, 8 etc) will always appear on this list since they were never visited. However, the real

question is *why* they were never visited.

If we perform Dijkstra's algorithm on the original weight matrix and find the fastest route form node 13 to 52, we find this:

```
print(Dijkst(12,51,weight_matrix))
```

Figure 3.7: Dijkstra's algorithm path from 13 to 52 (python indices)

Figure 3.8: Dijkstra's algorithm path from 13 to 52 with constant weight matrix

note that these are the python indices and not the actual node numbers.

If we look at figure 3.8, this means without taking the effect of congestion, the fastest path throughout the city is almost completely at the bottom of the map, by the river. This means that it is unlikely for a car to need to travel through the upper west side of the city, such as through nodes 1,3,5 and 2. The only way the shortest path for a car could include such nodes is if the rest of the city is *so* congested that the shortest path will indeed require these upper left nodes. We can see that edges like [3,2], [8,9] are not optimal for this reason, and are thus unused.

There are lots of edges that although aren't facing in the wrong direction in both the x and y direction, but will obviously never lead to an optimal solution. Such as [57,55] and [56,54]. This is because these nodes have a very small degree, 2 or 3. So if the optimal path runs through them (in the opposite direction), the other direction which tracks backwards will not be part of an optimal solution.

We can also consider the fact that if an edge is used by cars, the inverse of that edge is unlikely to be used again. This makes sense if we consider that 71 edges are unused out of a total of 156 edges (almost 1/2). So if we compare all the edges available to use and the edges unused, we should find a high correlation between pairs of the form [a,b] and [b,a] existing in either set. This is another reason highly used edges like [55,57] and [56,54] are not traversed in the opposite direction, as discussed in the paragraph above.

4. **What flow pattern do we observe for parameter $\epsilon = 0$?**

   See the *solution_epsilon0.py* file. The main program is exactly the same, but for one change. I have replaced epsilon = 0.01 with epsilon = float(0) where we update the weight matrix in the main for loop.

```
temp_wei = update_weight_matrix(float(0), cars_at_node, weight_matrix)
```

   Now, if we consider the formula for the adjusted weight matrix at each iteration, where $w_{ij} = w_{ij}^{(0)} + \epsilon \frac{c_i + c_j}{2}$, we see that the weight matrix remains unchanged.

As discussed in lectures, Dijkstra's algorithm is an example of a greedy algorithm. Since the time discrete process we are modelling uses Dijkstra's algorithm one node at a time (with the same weight matrix), we know that the shortest path from node 13 to node 52 will remain constant throughout the process.

Furthermore, let P be a node on the shortest path between node 13 and 52,

$$13 \rightarrow .. \rightarrow P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$$

Then the shortest path from node P to node 52 is

$$P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$$

Since the weight matrix is unchanged. If the shortest path between node P and node 52 were anything else then the shortest path between node 13 and node 52 would not be $13 \rightarrow ... \rightarrow P \rightarrow p_1 \rightarrow ... \rightarrow p_{n-1} \rightarrow 52$.

In other words, finding the shortest path from node to node with a constant weight matrix is the same as finding the overall shortest path using the same weight matrix. We can simply use the utilised edges to see that the flow of cars in the system follows this path. Printing the cars_at_node vector throughout the 200 minutes verifies this.

```
1  # regular dijkstra's path
2  print('the Dijkstra\'s path is ')
3  print(dijk.Dijkst(12, 51, weight_matrix))
4
5  utilised_edges = [[i, j] for i in range(noNodes)
6                    for j in range(noNodes)
7                    if edge_utilised[i, j]] # this matrix is defined in the m
8                                                #code
9  print('the utilised edges are')
10 print(utilised_edges)
```

Figure 3.9: Dijkstra's path with utilised edges output (both in Python indices for clarity)

Since all of these edges have a distinct head to tail connection, I can be sure the cars follow the path in the order we expect.

5. **An accident occurs at node 30 (python-index 29) which blocks any route to or from node 30. Which nodes are now the most congested and what is their maximum load? Which nodes (besides node 30) decrease the most in peak value, which nodes in- crease the most in peak value?**

   See *solution_accident_occurs.py*. The main code is exactly the same as before, but for these changes:

Since no car can reach node 30, we need to make the 30th row and 30th column of the weight matrix equal to 0s.

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

# The accident at node 30 means that the 30th row and 30th column is all 0
weight_matrix[29, :] = np.zeros(58, dtype=float)
weight_matrix[:, 29] = np.zeros(58, dtype=float)

temp_wei = weight_matrix.copy()
```

Once we begin the for loop iterating through the 200 minutes, we compute the next nodes as before. But there is no path between node 30 and 52, so we add this "if node!=29" statement to the next_nodes calculation.

We also insert a 0 at index 29, so the nodes align properly again.

```
# 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) if node != 29]

    next_nodes.insert(29, 29)  # send the 0 cars from 29 to itself
```

We can ignore node 30 when moving the cars through the system, so we add this else if statement.

```
 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))
            # really, we can just ignore node 30.
        elif j_node != 29:
```

Now we create this variable max_index_tracker_no30, which is simply the array of all the nodes with the maximum load they carry over the 200 iterations. This is then imported into the *solution_final.py* file as outlined before.

```
# Find the top 5 most congested nodes.
```

```
max_index_tracker_no30 = [[node+1, max_cars_at_node[node]]
                          for node in range(total_nodes)]
```

Now, back in *solution_final.py* we compare and print the required values.

```
top_eight = sorted(max_index_tracker_no30,
                   key=lambda node_and_max: -1 * node_and_max[1])[:8]
print(top_eight)
```

These are the top 8 most congested nodes when node 30 is blocked.

Figure 3.10: Top 8 most congested nodes with maximum loads when node 30 is blocked

Now we find the nodes which increase/decrease the most in peak value.

```
    differences = []
    for k in range(total_nodes):
        if k == 29:
            differences.append([k+1, 0])  # ignore when analysing
        else:
            differences.append([k+1, max_index_tracker[k][1]
                                - max_index_tracker_no30[k][1]])

    sorted_differences_most = \
        sorted(differences,
               key=lambda node_and_max: -1 * node_and_max[1])[:8]
    sorted_differences_least = \
        sorted(differences,
               key=lambda node_and_max: node_and_max[1])[:8]
    print(sorted_differences_most)
    print(sorted_differences_least)
```

Figure 3.11: The top 8 nodes which increased the most in peak value, followed by (next line) the top 8 nodes that decreased the most in peak value, all with their difference in peak values.

These elements represents the node number and the difference in maximum load from before to when node 30 is congested. So for example, node 6 increased its peak value by 5, and node 43 decreased its peak value by 14.