IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

M3SC Project 2

Changmin Yu

CID:00927624

March 21, 2017

1. PREPARATIONS

In this project, we are dealing with tasks scheduling, where dependencies and parallel execution need to be followed as instructed in the project. Firstly, we need to try to construct a file containing the information about the tasks, including the start and finish nodes, the virtual start and finish nodes and the dependencies as instructed in the project, and this file should have similar format as the "RomeEdges" file from Project 1. Note that in this project, I do not need two files containing the coordinates and the speeds between nodes to construct the weight matrix, here the weights between any two nodes is much simpler to compute. The procedure is shown as following:

1.1. "EDGES" FILE

Firstly I am going to construct a directed graph with 28 nodes to represent the flow, where there are 13 "start" nodes correspond to the starting points of tasks 0-12, nodes 0-12, 13 "finish" nodes correspond to the finishing points of tasks 0-12, nodes 13-25, a virtual start node that connects to all jobs' "start" nodes, node 26, and a virtual finish node that connects to all jobs' "finish" nodes, node 27.
Secondly, I need to define the weights of the edges between the nodes. The edge from the start node to the finish node of each job have weight equals to the duration of the corresponding job, i.e, 0->13 has weight 41, 1->14 has weight 51 and so on. And in order to have

the dependencies make sense, I define the connections from the finish node of one job to the start node of the next have an edge weight of 0.01 with respect to the dependency table 1.1 from the project, i.e, 13->1 has weight 0.01, 13->7 has weight 0.01 and so on. I use the similar definitions for the weights of the edges from the virtual start node to the start nodes of the jobs and the edges from the finish nodes of the jobs to the virtual finish node, i.e, 26->0 has weight 0.01, 13->27 has weight 0.01 and so on.
Now finally, I can construct my "Edges" file using TextEdit of the similar format of the file "RomeEdges". I will report below the first two rows of each categories I mentioned in the above paragraph, and I will report the entire file in the appendix at the end of the project.

```
0,  13, 41
1,  14, 51   % start->finish
26, 0,  0.01
26, 1,  0.01 % virtual start->start
13, 27, 0.01
14, 27, 0.01 % finish->virtual finish
13, 1,  0.01
13, 7,  0.01 % dependency relation
```

1.2. Import Modules

Before writing the codes for the project, I first need to import a few modules as following:

```
1  import numpy as np
2  import csv
3  import sys
```

1.3. Calculating the weight matrix

After constructing the file, I now need to write a function to calculate the weight matrix of the directed graph using the information extracted from the "Edges" file.

```
1  def calcWei(Ns,Nf,Dur):
2      # calculate the weight matrix of the nodes in the
3      # directed graph using the information extracted from the file
4      n = 28
5      # define the number of nodes in the graph
6      wei = np.zeros((n,n),dtype=float)
7      # initialise the weight matrix
8      for i in range(len(Dur)):
9      # allocate the weights to their corresponding position
10     # in the weight matrix
11         wei[int(Ns[i]),int(Nf[i])] = Dur[i]
12     return wei
```

1.4. BELLMAN-FORD ALGORITHM

In this project, I am going to use the Bellman-Ford Algorithm to compute the paths from the virtual start node to the virtual finish node using the weight matrix calculated using the above calcWei function and the "Edges" file. The reason I am choosing Bellman-Ford over Dijkstra will be demonstrated in the next section. Here I report the Bellman-Ford Algorithm below:

```
1   def BellmanFord(ist,isp,wei):
2     #--------------------------------
3     #  ist:    index of starting node
4     #  isp:    index of stopping node
5     #  wei:    adjacency matrix (V x V)
6     #
7     #  shpath: shortest path
8     #--------------------------------
9
10    V = wei.shape[1]
11
12    # step 1: initialization
13    Inf    = sys.maxint
14    d      = np.ones((V),float)*np.inf
15    p      = np.zeros((V),int)*Inf
16    d[ist] = 0
17
18    # step 2: iterative relaxation
19    for i in range(0,V-1):
20        for u in range(0,V):
21            for v in range(0,V):
22                w = wei[u,v]
23                if (w != 0):
24                    if (d[u]+w < d[v]):
25                        d[v] = d[u] + w
26                        p[v] = u
27
28    # step 3: check for negative-weight cycles
29    for u in range(0,V):
30        for v in range(0,V):
31            w = wei[u,v]
32            if (w != 0):
33                if (d[u]+w < d[v]):
34                    print('graph contains a negative-weight cycle')
35
36    # step 4: determine the shortest path
37    shpath = [isp]
```

```
38      while p[isp] != ist:
39          shpath.append(p[isp])
40          isp = p[isp]
41      shpath.append(ist)
42
43      return shpath[::-1]
```

2. Main Function

In this section I will report the main function I use in completing the project in parts along with the detailed demonstrations of each part.

Here I will explain the reason why I choose to use Bellman-Ford algorithm in section 1.4. Note that both Bellman-Ford algorithm and Dijkstra's algorithm are aimed to compute the shortest path between two nodes in the graph, but in this project, I need to compute the longest path from the virtual start node to the virtual finish node and then iteratively compute the remaining shorter job sequences. The original weight matrix of the graph is denoted by "wei", in this case, we can apply an algorithm that searches the shortest path to -wei to obtain the longest path. As we learnt from the lecture, Dijkstra's algorithm cannot be applied to graphs with negative weights, but Bellman-Ford algorithm can. Hence, despite Bellman-Ford algorithm is actually slower than the Dijkstra's algorithm, we have to use it in this project.

Now I will explain the code in detail.
Firstly, I need to import the data from the "Edges" file and store them in the lists "N_str", "N_fin" and Dur which are the lists of the start nodes, finish nodes, and their corresponding weights respectively, then apply the function "calcWei" to these lists to calculate the initial weight matrix of the directed graph, wei.

```
1  if __name__ == "__main__":
2
3      N_str = np.empty(0,dtype=float)
4      N_fin = np.empty(0,dtype=float)
5      Dur = np.empty(0,dtype=float)
6      with open('Edges','r') as file:
7          AAA = csv.reader(file)
8          for row in AAA:
9              N_str = np.concatenate((N_str,[int(row[0])]))
10             N_fin = np.concatenate((N_fin,[int(row[1])]))
11             Dur = np.concatenate((Dur,[float(row[2])]))
12     file.close()
13     # import the data from the "Edges" file.
14
```

```
    wei = calcWei(N_str,N_fin,Dur)
    # compute the initial weight matrix for the directed graph
```

Secondly, I apply BellmanFord to the virtual start and finish nodes and -wei to obtain the longest path through this graph. Then I compute the total duration of the longest path using a for loop. Note that I use range(0,L,2) to iterate since the start and finish nodes are consecutive, hence we need to obtain p and q, the start and finish nodes of each job in the path and then the corresponding duration of the job and add it to total duration of the path.

```
ist = 26
isp = 27
# Define the indices of the virtual start and finish node

lp = BellmanFord(ist,isp,-wei)
# using B-F algorithm to compute the longest path from the
# virtual start node to the virtual finish node
lp_new=lp[1:-1]
# get the path free of the virtual start and finish node
L = len(lp_new)-1
time = 0
for i in range(0,L,2):
    # calculating the duration of the longest path
    p=N_str==lp_new[i]
    q=N_fin==lp_new[i+1]
    time += int(Dur[p&q])
```

The next step is to iteratively determine the remaining shorter job sequences, until I have determined the start and finish times of all jobs. To do this, I firstly generate empty lists V, P and tp which are the lists of visited nodes, longest paths from the iterations and their corresponding durations respectively. The iterations terminate after we have visited all start nodes, i.e, we have executed all jobs, this can be done by using a while loop. Then I apply BellmanFord to all unvisited nodes and compute their durations using the method I used when calculating the duration of the longest path through the graph introduced in the above paragraph earlier and store them in the vector T, then I use T.argmax() to obtain the duration of the longest path in this iteration, compute the longest path, add this path to the list P and add its duration to the list tp. The final step of each iteration is to update the weight matrix by setting the weight from the start node to the finish node of each job in the path to zero and add the start nodes of the jobs in the path to the list of visited nodes, V.

```
# Initialisations
V = []
# the list of visited nodes, updated after each loop below P = []
# the list of the paths computed below
tp = []
```

```
# the list of the durations of the iterated longest paths computed below
while len(V)!=13:
    # the iteration terminates after we have visited all nodes
    T = np.zeros(13,dtype=int)
    # list of the durations of all paths computed
    for i in range(13):
        # compute the paths from every unvisted start nodes
        # then compute their durations and store them in the list T
        if i in V:
            pass
        else:
            path = BellmanFord(i,isp,-wei)
            l = len(path)-1
            t = 0
            for j in range(0,l,2):
                p=N_str==path[j]
                q=N_fin==path[j+1]
                # obtaining the job executed in the path,
                # p and q are the corresponding start and
                # finish node of the job
                t += Dur[p&q]
                # adding the duration of the job to the
                # duration of the path
        T[i] = t

    m=T.argmax()
    # obtain the index of the longest path in this iteration
    tp.append(T[m])
    # add the duration of the longest path in this
    # iteration to the list tp
    path2=BellmanFord(m,isp,-wei)
    P.append(path2)
    # add the longest path in this iteration to the list P

for k in range(0,len(path2)-1,2):
    # update the weight matrix by setting the weights of the edges
    # from the start nodes to the finish nodes of the jobs
    # in the longest path in this iteration and adding the
    # start nodes of the jobs in the longest path in this
    # iteration to the list of visited nodes, V
    wei[int(path2[k]), int(path2[k+1])] = 0
    V.append(path2[k])
```

The following step is to identify any possible combination of the shorter paths computed,

this can be done if it is allowed by the dependency table and the sum of the durations of the combination of the shorter paths is smaller than the duration of the longest path through the graph computed above given the initial computation is given.
The list "P" calculated previously is consist of the virtual start and finish nodes, start and finish nodes of the jobs, to obtain a clear visualisation of which jobs are executed in each sequence, I will use the following code to do so:

```
    P = [P[i][:-1] for i in range(len(P))]
    P_new = np.zeros((1, 6))
    # the new P list consist of the sequence of the jobs
    for i in range(len(P)):
        p1 = P[i]
        l = int(len(p1)/2)
        p2 = []
        if l != 0:
            for j in range(l):
                p2 += [p1[2*j]]
        else:
            p2 = [p1[0]]
        P_new.append(p2)
    P_new = P_new[1:]
```

Now I am going to compute a list of jobs that need to be processed before more than one other jobs, a dictionary with the information about the jobs with multiple dependencies, and a list of initial computations of the optimal start time of the 13 jobs. Note that a special case for doing this is that in this dictionary of information of nodes with multiple dependencies, the node 1 cannot be executed before job 0 is finished, hence, we need to take special care for this by adding an additional value equals to the duration of job 0.
The code for doing this is as following:

```
dur = [41,51,50,36,38,45,21,32,32,49,30,19,26]
# list of durations of the jobs
IS = np.zeros(13,dtype=int)
for i in range(len(C1)):
    c = C1[i]
    # the list of nodes need to be processed before
    # multiple other jobs
    d =dic[c]
    # compute a dictionary of the information of the nodes
    # with multiple dependencies
    for j in range(len(P_new)):
        p = P_new[j]
        if c in p:
            ind = p.index(c)
            d.remove(p[ind+1])
```

```
for c in C1:
    if c != 1:
        # solve the issue for the case when the job with
        # multiple dependency is job 1
        d = dic[c]
        for i in range(len(d)):
            IS[d[i]] += dur[c]
    else:
        # add the duration of job 0
        IS[12] = 41 + dur[c]
```

Then I will identify any combination possible to reduce the number of processes executed in parallel. After initial computation and compare with the dependency table, I make the following combination and update the "P_new" list and the "tp" list of time of the paths:

```
    P_new[4] = P_new[5]+P_new[4]
    P_new = P_new[:-1]
    P[4]= P[5]+P[4]
    P = P[:-1]
    tp[4] += tp[5]
    tp = tp[:-1]
```

Next, I compute the list of optimal start times of the jobs after updating the paths:

```
    S = np.zeros(13,dtype=int)
    for p in P_new:
        S[p[0]] = 0
        if len(p) == 2:
            S[p[1]] = dur[p[0]]
        elif len(p) == 3:
            S[p[1]] = dur[p[0]]
            S[p[2]] = S[p[1]] + dur[p[1]]
        else:
            pass
```

Finally, I generate a final list of optimal start times using the "IS" and "S" lists computed above by taking the maximum of the entries of the same indices from both lists, i.e, SS[i] = max(IS[i],S[i]) $\forall i$. Next add the duration of job i to the ith entry of the list "SS" computed to obtain a list "F" which is the optimal finish times of the 13 jobs. Note that one special is job 11, since it cannot be considered by the above codes yet it cannot start at time 0, hence I need to manually add the duration of job 6 to the start time of job 11.

```
SS = [max(S[i],IS[i]) for i in range(13)]
SS[11] += 21 # special case job 11
# compute the final optimal start times list
F = np.zeros(13, dtype=int)
F = [SS[i] + dur[i] for i in range(13)]
```

```
# compute the optimal finish times list
```

The final step is print our results, the longest path, the shorter paths using iterative method as well as in the form of the sequence of jobs, and the optimal start and finish times of the jobs 0 to 12.

```
# report the outcome
print 'The longest path through this graph from the
       virtual start node to the virtual finish node
       is',lp,'with duration',time
print 'The list of the longest path and the remaining shorted paths:'
for i in range(len(tp)):
    print 'The path',[ist]+P[i],'where expressed in the
           form of strings of jobs is:', P_new[i]
print 'The number of processes in parallel is',len(tp)
for i in range(13):
    print 'The optimal startand finish time for
           job',i,'are',SS[i],'and',F[i],'respectively.'
```

3. Results

3.1. Longest Path

The longest path through this graph from the virtual start node to the virtual finish node is reported as following:

The longest path through this graph from the virtual start node to the virtual finish node is [26, 0, 13, 1, 14, 4, 17, 27] with duration 130.

If we rewrite this in a string of jobs, it is:
virtual start -> 0 -> 1 -> 4 -> virtual finish

3.2. The remaining job sequences

The remaining shorter job sequences until I have determined the start and finish times of all jobs (visited all jobs) and their corresponding durations is reported as following (here I will report the longest path as well) and the number of processes that can be executed in parallel:

The list of the longest path and the remaining shorted paths:
The path [26, 0, 13, 1, 14, 4, 17] where expressed in the
form of strings of jobs is: [0, 1, 4]
The path [26, 6, 19, 5, 18, 7, 20] where expressed in the
form of strings of jobs is: [6, 5, 7]
The path [26, 2, 15, 3, 16] where expressed in the form of

strings of jobs is: [2, 3]
The path [26, 9, 22, 11, 24] where expressed in the form of strings of jobs is: [9, 11]
The path [26, 8, 21, 10, 23, 12, 25] where expressed in the form of strings of jobs is: [8, 10, 12]
The number of processes in parallel is 5

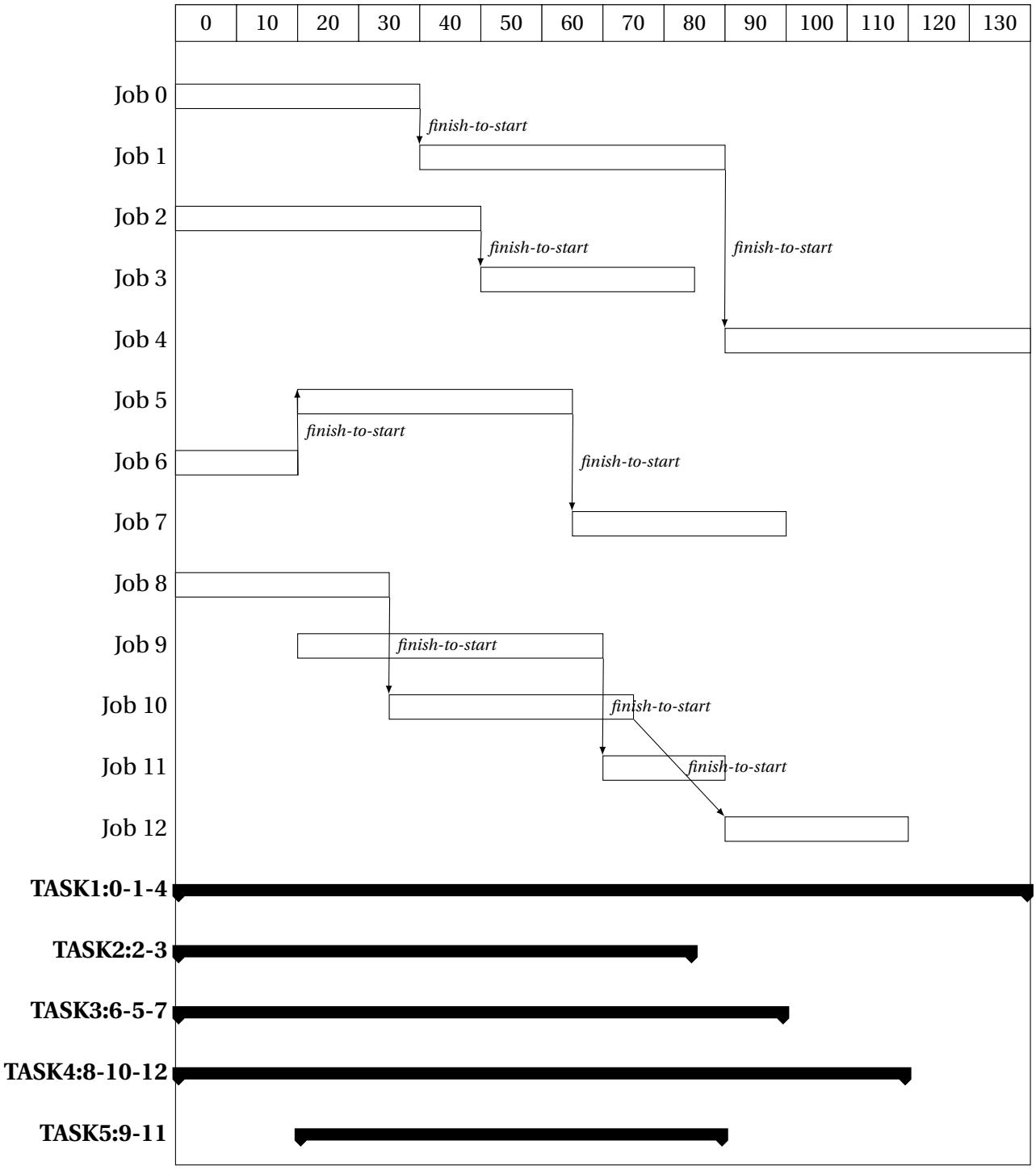
3.3. New Table with updated information

We can determine the optimal start and finish time for each job, the list of start and stop times for all jobs in the work-flow along with the original table is reported below:

job	duration	has to be completed before	optimal start time	optimal finish time
0	41	1, 7, 10	0	41
1	51	4, 12	41	92
2	50	3	0	50
3	36		50	86
4	38		92	130
5	45	7	21	66
6	21	5, 9	0	21
7	32		66	98
8	32		0	32
9	49	11	21	70
10	30	12	41	71
11	19		70	89
12	26		92	118

3.4. VISUALISATION OF THE WORKFLOW IN FORM OF A GANTT CHART

The visualisation in form of a Gantt chart is shown below:



As can be seen from the Gantt chart, 5 tasks can be performed in parallel, where the 5 tasks are the strings of jobs reported at the end of 3.2.

Appendices

A. "Edges" file

```
0,  13, 41
1,  14, 51
2,  15, 50
3,  16, 36
4,  17, 38
5,  18, 45
6,  19, 21
7,  20, 32
8,  21, 32
9,  22, 49
10, 23, 30
11, 24, 19
12, 25, 26
26, 0,  0.01
26, 1,  0.01
26, 2,  0.01
26, 3,  0.01
26, 4,  0.01
26, 5,  0.01
26, 6,  0.01
26, 7,  0.01
26, 8,  0.01
26, 9,  0.01
26, 10, 0.01
26, 11, 0.01
26, 12, 0.01
13, 27, 0.01
14, 27, 0.01
15, 27, 0.01
16, 27, 0.01
17, 27, 0.01
18, 27, 0.01
19, 27, 0.01
20, 27, 0.01
21, 27, 0.01
22, 27, 0.01
23, 27, 0.01
24, 27, 0.01
25, 27, 0.01
```

```
13, 1,  0.01
13, 7,  0.01
13, 10, 0.01
14, 4,  0.01
14, 12, 0.01
15, 3,  0.01
18, 7,  0.01
19, 5,  0.01
19, 9,  0.01
22, 11, 0.01
23, 12, 0.01
```

B. Complete Codes

```python
import numpy as np
import csv
import sys

def BellmanFord(ist,isp,wei):
    #---------------------------------
    #  ist:    index of starting node
    #  isp:    index of stopping node
    #  wei:    adjacency matrix (V x V)
    #
    #  shpath: shortest path
    #---------------------------------

    V = wei.shape[1]

    # step 1: initialization
    Inf    = sys.maxint
    d      = np.ones((V),float)*np.inf
    p      = np.zeros((V),int)*Inf
    d[ist] = 0

    # 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 != 0):
                    if (d[u]+w < d[v]):
                        d[v] = d[u] + w
                        p[v] = u
```

```
31
32      # step 3: check for negative-weight cycles
33      for u in range(0,V):
34          for v in range(0,V):
35              w = wei[u,v]
36              if (w != 0):
37                  if (d[u]+w < d[v]):
38                      print('graph contains a negative-weight cycle')
39
40      # step 4: determine the shortest path
41      shpath = [isp]
42      while p[isp] != ist:
43          shpath.append(p[isp])
44          isp = p[isp]
45      shpath.append(ist)
46
47      return shpath[::-1]
48
49  def calcWei(Ns,Nf,Dur):
50      # calculate the weight matrix of the nodes in the
51      # directed graph using the information extracted from the file
52      n = 28
53      # Define the number of nodes in the graph
54      wei = np.zeros((n,n),dtype=float)
55      # initialise the weight matrix
56      for i in range(len(Dur)):
57      # allocate the weights to their corresponding position
58      # in the weight matrix
59          wei[int(Ns[i]),int(Nf[i])] = Dur[i]
60      return wei
61
62  if __name__ == "__main__":
63
64      N_str = np.empty(0,dtype=float)
65      N_fin = np.empty(0,dtype=float)
66      Dur = np.empty(0,dtype=float)
67      with open('Edges','r') as file:
68          AAA = csv.reader(file)
69          for row in AAA:
70              N_str = np.concatenate((N_str,[int(row[0])]))
71              N_fin = np.concatenate((N_fin,[int(row[1])]))
72              Dur = np.concatenate((Dur,[float(row[2])]))
73      file.close()
74      # import the data from the "Edges" file.
```

```
75
76      wei = calcWei(N_str,N_fin,Dur)
77      # compute the initial weight matrix for the directed graph
78      ist = 26
79      isp = 27
80      # Define the indices of the virtual start and finish node
81
82      lp = BellmanFord(ist,isp,-wei)
83      # using B-F algorithm to compute the longest path from
84      # the virtual start node to the virtual finish node
85      lp_new=lp[1:-1]
86      # get the path free of the virtual start and finish node
87      L = len(lp_new)-1
88      time = 0
89      for i in range(0,L,2):
90          # calculating the duration of the longest path
91          p=N_str==lp_new[i]
92          q=N_fin==lp_new[i+1]
93          time += int(Dur[p&q])
94
95      # Initialisations
96      V = []
97      # the list of visited nodes, updated after each loop below
98      P = []
99      # the list of the paths computed below
100     tp = []
101     # the list of the durations of the iterated longest
102     # paths computed below
103     while len(V)!=13:
104         # the iteration terminates after we have visited all nodes
105         T = np.zeros(13,dtype=int)
106         # list of the durations of all paths computed
107         for i in range(13):
108             # compute the paths from every unvisted start
109             # nodes then compute their durations and store
110             # them in the list T
111             if i in V:
112                 pass
113             else:
114                 path = BellmanFord(i,isp,-wei)
115                 l = len(path)-1
116                 t = 0
117                 for j in range(0,l,2):
118                     p=N_str==path[j]
```

```
                    q=N_fin==path[j+1]
                    # obtaining the job executed in the
                    # path, p and q are the corresponding
                    # start and finish node of the job
                    t += Dur[p&q]
                    # adding the duration of the job to
                    # the duration of the path
            T[i] = t

        m=T.argmax()
        # obtain the index of the longest path in this iteration
        tp.append(T[m])
        # add the duration of the longest path in this
        # iteration to the list tp
        path2=BellmanFord(m,isp,-wei)
        P.append(path2)
        # add the longest path in this iteration to the list P

        for k in range(0,len(path2)-1,2):
            # update the weight matrix by setting the
            # weights of the edges from the start nodes to
            # the finish nodes of the jobs in the longest
            # path in this iteration and adding the start
            # nodes of the jobs in the longest path in
            # this iteration to the list of visited nodes, V
            wei[int(path2[k]), int(path2[k+1])] = 0
            V.append(path2[k])

    P = [P[i][:-1] for i in range(len(P))]
    P_new = [np.zeros((1, 6))]
    # the new P list consist of the sequence of the jobs
    for i in range(len(P)):
        p1 = P[i]
        l = int(len(p1)/2)
        p2 = []
        if l != 0:
            for j in range(l):
                p2 += [p1[2*j]]
        else:
            p2 = [p1[0]]
        P_new.append(p2)
    P_new = P_new[1:]

    dur = [41,51,50,36,38,45,21,32,32,49,30,19,26]
```

```
    # list of durations of the jobs
    IS = np.zeros(13,dtype=int)
    for i in range(len(C1)):
        c = C1[i]
        # the list of nodes need to be processed before
        # multiple other jobs
        d =dic[c]
        # compute a dictionary of the information of the
        # nodes with multiple dependencies
        for j in range(len(P_new)):
            p = P_new[j]
            if c in p:
                ind = p.index(c)
                d.remove(p[ind+1])
    for c in C1:
        if c != 1:
            # solve the issue for the case when the job
            # with multiple dependency is job 1
            d = dic[c]
            for i in range(len(d)):
                IS[d[i]] += dur[c]
        else:
            # add the duration of job 0
            IS[12] = 41 + dur[c]


    P_new[4] = P_new[5]+P_new[4]
    P_new = P_new[:-1]
    P[4]= P[5]+P[4]
    P = P[:-1]
    tp[4] += tp[5]
    tp = tp[:-1]

    S = np.zeros(13,dtype=int)
    for p in P_new:
        S[p[0]] = 0
        if len(p) == 2:
            S[p[1]] = dur[p[0]]
        elif len(p) == 3:
            S[p[1]] = dur[p[0]]
            S[p[2]] = S[p[1]] + dur[p[1]]
        else:
            pass

```

```
207
208     SS = [max(S[i],IS[i]) for i in range(13)]
209     SS[11] += 21 # special case job 11
210     # compute the final optimal start times list
211     F = np.zeros(13, dtype=int)
212     F = [SS[i] + dur[i] for i in range(13)]
213     # compute the optimal finish times list
214
215     # report the outcome
216     print 'The longest path through this graph from the
217            virtual start node to the virtual finish node
218            is',lp,'with duration',time
219     print 'The list of the longest path and the remaining
220            shorted paths:'
221     for i in range(len(P)):
222         print 'The path',[ist]+P[i],'where expressed in
223                the form of strings of jobs is:', P_new[i]
224     print 'The number of processes in parallel is',len(tp)
225     for i in range(13):
226         print 'The optimal startand finish time for
227                job',i,'are',SS[i],'and',F[i],'respectively.'
```

——————————————————END OF PROJECT 2 ——————————————————-