IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC) - Project 2

Isa Majothi

March 23, 2017

1 APPROACH AND SOLUTION STRATEGY

To design a workflow and job schedule that minimises the duration of a process which involves the completion of numerous jobs, some of which are dependent upon the completion of others, I took the given function **BellmanFord** and modified it slightly so that it could iteratively determine the earliest possible start times (and therefore earliest finish times given the job duration times) for each job, calling my variation of the function **MyBellmanFord**. Since starting from the longest string of jobs and iteratively determining the subsequent shorter job strings from this leads us to these earliest times, I used **MyBellmanFord** by applying it to the negative weight matrix of our directed graph so that it would determine the "shortest path" through the negative network, therefore giving us the longest string of jobs, and from this I iteratively determined the earliest start and finish times for each job. From this information, an optimal workflow can be determined and visualised in the form of a Gantt chart, which tells us how we can efficiently execute as many jobs in parallel as is possible whilst still respecting the dependencies of some jobs on others; for example, since job 2 must be completed before job 3 we can't have job 3 starting before the completion of job 2 in our workflow, but since job 8 is independent of both jobs 2 and 3 (and every other job in fact), it can be completed simultaneously to any of these jobs.

As instructed by the given algorithm, to determine the optimal workflow I introduced a "start" node and a "finish" node for each of the 13 jobs, where the weight of the edge between the "start" node of a job and its respective "finish" node was the time taken for the job to be completed. I then introduced one "virtual" start node and one "virtual" finish node, which were connected *to* all job's "start" nodes and *from* all job's "finish" nodes respectively; these edges were all of weight 0. Where there were dependencies, I connected the "finish" node of a job to the "start" node of a job that is dependent upon the first job with an edge of weight zero e.g. job 11 is dependent upon job 9, so the edge between the "finish" node

of job 9 and the “start” node of job 11 had weight 0. All other entries were then set to a relatively large negative number called *noLink1*, details of which can be found in the code in Section 2. It is also worth noting that since we have 13 jobs to complete, each of which has a “start” and “finish” node, as well as one "virtual" start node and one "virtual" finish node, our directed graph is represented by a 28 by 28 matrix, which is *weightMatrix* in Section 2.

An advantage of using the Bellman-Ford algorithm is that it can be applied to directed graphs consisting of negative weights, whilst Dijkstra’s algorithm can’t consider negative weights; in order to determine the first longest string of jobs, I ran **MyBellmanFord** with the negative weight matrix, namely -*weightMatrix*, and **MyBellmanFord** is programmed so that it takes |noLink1| as an argument so that any edges of weight |noLink1| in -*weightMatrix* are never used. Since **MyBellmanFord** is designed to find the shortest route through the network whilst respecting the dependencies, setting the start point to the "virtual" start node and the end point to the "virtual" finish node, **MyBellmanFord** determines the “shortest path” of the negative times, which is hence the longest string of jobs for the actual times.

Once this current longest job string has been obtained, I ensure that any jobs appearing in this string are never included in another shorter job string unnecessarily by removing the edge between the "finish" nodes of these jobs and the "virtual" finish node; I do this by updating *weightMatrix* so that these edges now have weight *noLink1*. I then store this current longest string, which is called *longestString* in the code, in the list *workFlow*, which is a list of lists. Finally, with this updated version of *weightMatrix*, I run **MyBellmanFord** again to find the next longest string of jobs and store this in *workFlow*; the iterations terminate once every job has appeared in a longest job string at least once, because this means we have sufficient information to determine the earliest start and finish times for all jobs as we do not wish to run **MyBellmanFord** more times than is necessary because this is inefficient.

Once we have determined the earliest start and finish times for all jobs, which are stored in the arrays *earliestStartTimes* and *earliestFinishTimes*, we can start to consider the the most efficient way of running as many jobs in parallel as is possible whilst respecting the dependencies, namely the workflow such that we minimise the total time taken to complete all jobs, and from this then executes as many jobs as we can in parallel whilst minimising the number of parallel processes (which could possibly represent the number of production lines we require to complete all of the jobs). It is worth noting that the first list in *workFlow* is the longest job string, and this therefore gives us the minimum amount of time taken to complete all of the 13 jobs, and from there we try and complete jobs in parallel so that we do as few parallel processes as is possible but also do not exceed this shortest time. This optimal workflow is then visualised in the form of a Gantt chart.

One potential issue that can arise when using the Bellman-Ford algorithm is the existence of negative-weight cycles in the directed graph. Since I use -*weightMatrix*, which consists of negative weights, if there happened to be a negative cycle this would mean that in effect we would stay in this cycle forever as the total weight of the path would be continually decreasing because the Bellman-Ford algorithm minimises this total weight. However, in the context of jobs with dependencies it does not make sense for us to have a negative-weight cycle; this would correspond to a closed loop of jobs depending on others which in effect results in a job not being able to start until it has itself finished, which is of course contradictory. However, the Bellman-Ford algorithm is programmed to detect such cycles, and we have a directed acyclic graph so there is no danger of negative-weight cycles occurring.

2 PYTHON CODE TO FIND OPTIMAL JOB SCHEDULING

I have included the code that I used to optimise the job scheduling in this section below, splitting it into various subsections which allow the thinking behind my approach to be seen and understood step-by-step. The code is explained in ample detail at each stage, elaborating on from the ideas explained in Section 1.

2.1 MYBELLMANFORD FUNCTION

Below is my slightly modified version of the Bellman-Ford algorithm which I later use to find the longest strings of jobs. Since the template of this algorithm was provided to us, I have indicated which sections remained the same and where I made tailored modifications to suit the task at hand.

```
1  import numpy as np
2  import scipy as sp
3  import csv
4  import sys
5
6  #Using the BellmanFord function provided, I modified it by
7  #introducing a new argument noLink, which is a real number
8  #pre-determined based on the weights of the weight matrix,
9  #which in out case is based on the job times for any
10 #given list of jobs that we may be given
11 def MyBellmanFord(ist,isp,wei,noLink):
12
13     #ist = index of start node
14     #isp = index of end node
15     #wei = weight matrix whose weights in our case represent
16     #      the times between nodes
17     #noLink = real number which is unique to the
18     #         pre-constructed weight matrix, and is used below
19     #         so that we do not use any edges of weight noLink
20     #Note: noLink is a really big number which symbolises
21     #      there being no link between two nodes. In the
22     #      original BellmanFord function, an entry of 0
23     #      represented there being no connection between two
24     #      nodes, but in this case the weights represent
25     #      times and we do wish to consider time periods of
26     #      0 length, so now noLink represents a very large
27     #      time which is undesirable to us, hence we do not
28     #      use edges of this weight
29
30     #Assign V to be the number of nodes in our network
31     V = wei.shape[1]
32
```

```
33      #Step 1: Initialization (this is provided to us)
34      Inf    = sys.maxint
35      d      = np.ones((V),float)*np.inf
36      p      = np.zeros((V),int)*Inf
37      d[ist] = 0
38
39      #Step 2: Iterative relaxation (this was given to us)
40      #Note: One subtle change I made was on line 50 below,
41      #      namely if (w != noLink), so we only consider edges
42      #      where we do not encounter a place at which there is
43      #      no connection between two nodes, which is
44      #      represented by an edge of weight noLink in the
45      #      pre-constructed weight matrix
46      for i in range(0,V-1):
47          for u in range(0,V):
48              for v in range(0,V):
49                  w = wei[u,v]
50                  if (w != noLink):
51                      if (d[u]+w < d[v]):
52                          d[v] = d[u] + w
53                          p[v] = u
54
55      #Step 3: Check for negative-weight cycles (this was given
56      #        to us). I did make a small change on line 66,
57      #        namely if (w != noLink) as opposed to if (w != 0)
58      #Note: due to the fact that we have a directed acyclic
59      #      graph, we should not expect to find any
60      #      negative-weight cycles, but it must be considered
61      #      since the Bellman-Ford algorithm can take negative
62      #      weights
63      for u in range(0,V):
64          for v in range(0,V):
65              w = wei[u,v]
66              if (w != noLink):
67                  if (d[u]+w < d[v]):
68                      print('graph contains a negative-weight
69                                                          cycle')
70
71      #Step 4: Determine the shortest path (this was given to
72      #        us), and store it in the list shpath
73      shpath = [isp]
74      while p[isp] != ist:
75          shpath.append(p[isp])
76          isp = p[isp]
77      shpath.append(ist)
78
79      return shpath[::-1]
```

2.2 CONSTRUCT THE WEIGHT MATRIX

Below is the code I used to construct the weight matrix for the jobs that we had, which is called *weightMatrix*. I have explained below the process I used to ensure that *weightMatrix* was suitable regarding helping us to find the optimal solution. For completeness, I have also included at the bottom of this subsection the text file **JobInformation** (see page 7) which contains the information that was read into PYTHON in order to construct *weightMatrix*.

```
#calcWei is a function that was given to us, which produces
#a weight matrix given some information that is inputted as
#arguments of the function in the form of arrays
def calcWei(RA,RB,RT):

    #Set n to be 2*(number of jobs) + 2, where in our case the
    #number of jobs that need completing is 13, giving n = 28.
    #This is so that in our weight matrix, every job has a
    #"start" and "finish" node, and ensures the existence of a
    #"virtual" start node and a "virtual" finish node so we
    #can implement the given algorithm
    n = 28
    wei = np.zeros((n,n),dtype=float)
    m = len(RA)
    for i in range(m):
        wei[RA[i],RB[i]] = RT[i]
    return wei

#Read in the data containing the job duration times and job
#dependencies from the text file JobInformation, and store
#the data in the arrays startNodes, finishNodes and jobTimes
startNodes = np.empty(0,dtype=int)
finishNodes = np.empty(0,dtype=int)
jobTimes = np.empty(0,dtype=float)

with open('JobInformation','r') as file:
    AAA = csv.reader(file)
    for row in AAA:
        startNodes=np.concatenate((startNodes,[int(row[0])]))
        finishNodes=np.concatenate((finishNodes,[int(row[1])]))
        jobTimes=np.concatenate((jobTimes,[float(row[2])]))
file.close()

#Compute the weight matrix for our jobs using the information
#imported from JobInformation
weightMatrix = calcWei(startNodes,finishNodes,jobTimes)

#Assign maxTime to be the largest entry of weightMatrix, which
#gives us the time of the job with the longest duration time
maxTime = np.amax(weightMatrix)
```

```
#Assign noLink1 to be the -maxTime*100, which will be used
#by MyBellmanFord to determine where no edges exist between
#any two given nodes
#Note: For any given job list, the value of maxTime and
#      therefore noLink1 will be specific and always
#      applicable, so it is a valid way of assigning and
#      recognising where there is no connection between two
#      nodes in the directed graph
noLink1 = -maxTime*100

#Where we originally had a weight of 0 between two nodes in
#the original weightMatrix, due to the functionality of
#calcWei this first 0 symbolised the fact there is no
#connection, so replace all of these 0's by noLink1
index1 = np.where(weightMatrix == 0)
weightMatrix[index1] = noLink1

#Where we have a weight of -1 between two nodes, I used this
#to represent either a job dependency or an edge between the
#"virtual" start and finish nodes in the file JobInformation;
#since we need all of these edges to have weight 0, change all
#of these -1's to 0's. This must happen after the previous
#line so the two sets of 0's are not confused
index2 = np.where(weightMatrix == -1)
weightMatrix[index2] = 0
```

As mentioned in Section 1, I introduced a "start" node and a "finish" node for all jobs, as well as a "virtual" start and "virtual" finish node. For reference in the file **JobInformation**, for all 13 jobs I have included the PYTHON-indexes of these "start" and "finish" nodes in the table below. For convenience, I have not stated the PYTHON-indexes of the "virtual" nodes in the table; the PYTHON-index of the "virtual" start node is 26, and 27 for the "virtual" finish node. Note that the third column in **JobInformation** represents the first set of weights.

Job	"Start" node PYTHON-index	"Finish" node PYTHON-index
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15
8	16	17
9	18	19
10	20	21
11	22	23
12	24	25

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

2.3 ITERATIVELY DETERMINE THE WORKFLOW

After constructing *weightMatrix*, I negated it and inputted it into **MyBellmanFord**. The following code explains the iterative process, and I include the output of *workFlow*, which contains the longest list of jobs in descending order given by each iteration of **MyBellmanFord**.

```
#Initialise a list of jobs called jobList, which keeps track
#of which jobs we are yet to find in a current longest job
#sequence; once a job appears in a current longest string, it
#will be removed from jobList. This is to ensure that we do
#not  run the function MyBellmanFord unnecessarily, since this
#will be computationally expensive so we want to run the
#function as few times as possible
#Note: I used 13 since we have 13 jobs (numbered from 0 to 12)
#      to complete, but this can be changed accordingly
jobList = list(np.arange(13))

#Initialise workFlow to store the longest string of jobs
#before we then iteratively determine the subsequent shorter
#job strings
#Note: I will not store sub-sequences of any job strings that
#      are found to be the longest strings at any given
#      iteration e.g if [0,1,4] is found to be a longest
#      string, the subsequent job string [0,1] will not be
#      stored in workFlow, as this will lead to unnecessary
#      implementations of MyBellmanFord
workFlow = []

#Whilst we still need to determine the earliest possible start
#and finish times for a job, run MyBellmanFord using the
#current weightMatrix (this will be altered slightly below),
#with the "virtual" start node (index 26) as our starting node
#and the "virtual" finish node (index 27) as our final node
while len(jobList) != 0:

    #Obtain the current longest string of jobs, and store it
    #in the list longestString. Since we wish to find the
    #longest string, I input -weightMatrix because the
    #Bellman-Ford algorithm is designed to find the shortest
    #path, so by doing this it finds the shortest path for the
    #negative times which is hence the longest path for the
    #positive times.
    #Also recall that noLink1 is a negative number, but in
    #-weightMatrix any entries that were initially noLink1
    #now become equal to -noLink1, so I want MyBellmanFord to
    #recognise -noLink1 as there not being a connection
    #between two nodes in -weightMatrix as opposed to noLink1
    longestString = MyBellmanFord(26,27,-weightMatrix,-noLink1)
```

```
    #Since longestString contains the indexes of the "virtual"
    #start node, "virtual" finish node and the "start" and
    #"finish" nodes of all of the respective jobs in between,
    #the code below ensures that longestString consists of a
    #string of jobs with the correct numbering i.e. the
    #desired string of jobs which can easily be read and
    #respects the numbering of the jobs in question
    del longestString[0]
    del longestString[-1]
    longestString = longestString[1::2]

    for i in range(len(longestString)):
        longestString[i] = (longestString[i]-1)/2

    #Once we have obtained the longest string of jobs in its
    #desired form, add it to the list of strings workFlow
    #using append
    workFlow.append(longestString)

    #To ensure we only run MyBellmanFord as many times as is
    #absolutely necessary, once a job appears in a longest
    #string I remove the connection between the finish nodes
    #of these jobs and the "virtual" finish node; this is to
    #avoid the occurrence of sub-sequences of jobs appearing
    #in workFlow. I also remove these jobs from jobList
    #because we now know the order in which these jobs must
    #appear, meaning we do not run MyBellmanFord excessively
    #for the reasons outlined above
    for i in longestString:
        if i in jobList:
            jobList.remove(i)
            weightMatrix[2*i+1,27] = noLink1
```

After running this code, I obtained the following job sequences (after needing to run **MyBellmanFord** seven times) which are used in Section 2.4 to obtain the earliest possible start and finish times for all jobs. Note that there are no sub-sequences in *workFlow*, which was my aim based on making the iterative process as efficient as possible.

```
workFlow
= [ [0, 1, 4],  [0, 1, 12],  [6, 5, 7],  [6, 9, 11],  [2, 3],
      [0, 10],          [8] ]
```

2.4 Obtain the earliest start and finish times for each job

After having obtained the job sequences stored in *workFlow*, from these job sequences I can now obtain the earliest possible start times (and therefore finish times) for all jobs, so whilst respecting the dependencies no job can start or finish before these earliest times. The earliest start times are stored in the array *earliestStartTimes*, and the earliest finish times are stored in the array *earliestFinishTimes*.

```
#List the times taken to complete each of the jobs in the
#array jobDurations so that we can refer to it when
#determining the earliest possible start and finish times for
#the jobs from workFlow
jobDurations = [41,51,50,36,38,45,21,32,32,49,30,19,26]

#Initialise two arrays of size 13 since we have 13 jobs, which
#will store the earliest possible start and finish times for
#each of the jobs
earliestStartTimes = np.zeros(13)
earliestFinishTimes = np.zeros(13)

#For every string of jobs in workFlow, cumulatively determine
#the earliest possible start and finish times for all of the
#jobs by going through every job sequence in workFlow.
#Note: due to the fact that workFlow does not contain any
#      sub-sequences, as well as the fact workFlow begins
#      with the longest string and descends from there, even
#      if we do have the occurrence of a particular job in
#      multiple strings, the job orders will not conflict with
#      each other, meaning it is acceptable to override
#      values in the iterations below. This is because every
#      time we obtain a longest path, we know for a fact that
#      this must be the longest way of completing EVERY job in
#      that sequence, and it is not possible for the order to
#      change later on because if it did, then the original
#      longest string containing this job wouldn't actually be
#      the longest string.
for string in workFlow:

    #sumTime is initialised to 0, and as we go through each
    #string of jobs we add the duration time of the current
    #job to it so that it keeps track of the cumulative time
    #for a given string as we go through it
    sumTime = 0
    for i in string:
        earliestStartTimes[i] = sumTime
        sumTime = sumTime + jobDurations[i]
        earliestFinishTimes[i] = sumTime
```

The values stored in the arrays *earliestStartTimes* and *earliestFinishTimes* are included below; in addition to including the arrays, I have tabulated this information for convenience, so these earliest times can easily be seen.

```
earliestStartTimes
= array([   0.,  41.,   0.,  50.,  92.,  21.,   0.,  66.,
            0.,  21.,  41.,  70.,  92.])

earliestFinishTimes
= array([  41.,  92.,  50.,  86., 130.,  66.,  21.,  98.,
           32.,  70.,  71.,  89., 118.])
```

Job	Earliest start time	Earliest finish time
0	0	41
1	41	92
2	0	50
3	50	86
4	92	130
5	21	66
6	0	21
7	66	98
8	0	32
9	21	70
10	41	71
11	70	89
12	92	118

3 Optimise the workflow

After having obtained the earliest possible start and finish times for all of the 13 jobs, I visualised the workflow in the form of a Gantt chart because doing so enables us to determine which jobs can be performed in parallel with others so we can minimise the number of production lines needed to complete all of the jobs. Based on the earliest possible start and finish times, I constructed a Gantt chart (see Figure 3.1) in which each job starts and ends at its earliest possible time. The different colours indicate different production lines, so if we were to complete all 13 jobs for the times in the table on page 11, we would require a minimum of five production lines represented by the five different colours.

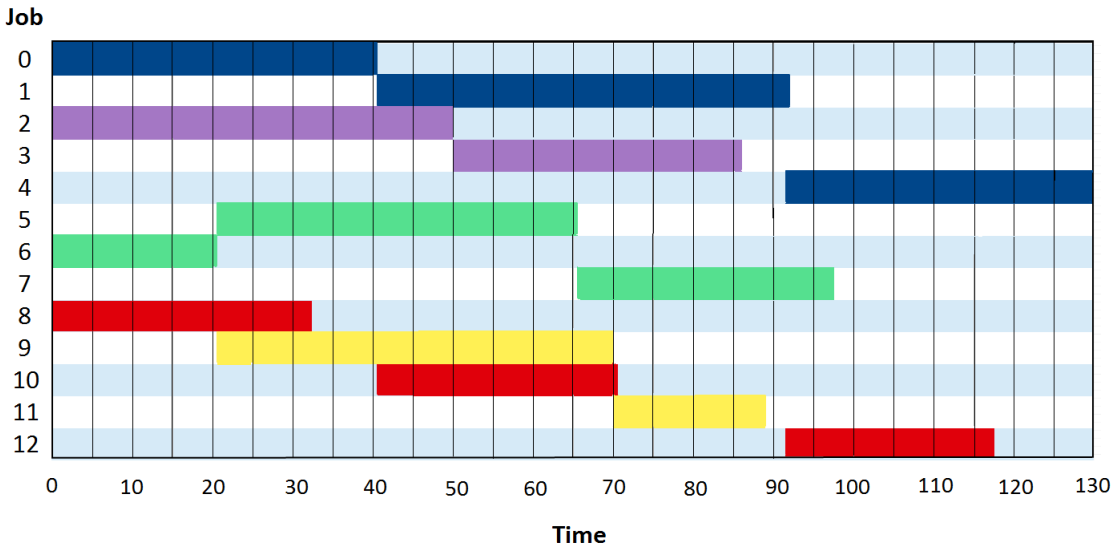

Figure 3.1: Gantt chart for which all jobs start and finish at the earliest possible times

One benefit of visualising the earliest start and finish times in a Gantt chart is that it allows us to, upon inspection, effectively slide jobs along to the right (never the left) so we can try and further reduce the number of production lines required. The times displayed in Figure 3.1 are the *earliest* start and finish times, so we can still start some jobs later than these times and yet finish them before the shortest possible time needed to complete all jobs, which was time 130 in this case since the longest job string was found to be [0,1,4] (see *workFlow* on page 9) which takes time 130 to complete.

After analysing Figure 3.1, I noticed that job 8 could be completed after job 7 (see green lines in Figure 3.1) at time 98 and still not finish later than time 130 (in fact it would finish at time 130). I also observed that job 10 can be started after job 11 (see yellow lines in Figure 3.1) at time 89, and that job 12 could still start at time 92 but could be completed by the same production line as the one completing jobs 2 and 3 (see purple lines in Figure 3.1). After having made these observations, I produced a second Gantt chart (see Figure 3.2) which only displays four colours, symbolising the fact that it is actually possible to complete all 13 jobs in the minimum total time of 130 with only four production lines, as opposed to the five production lines given when using the earliest possible start times.

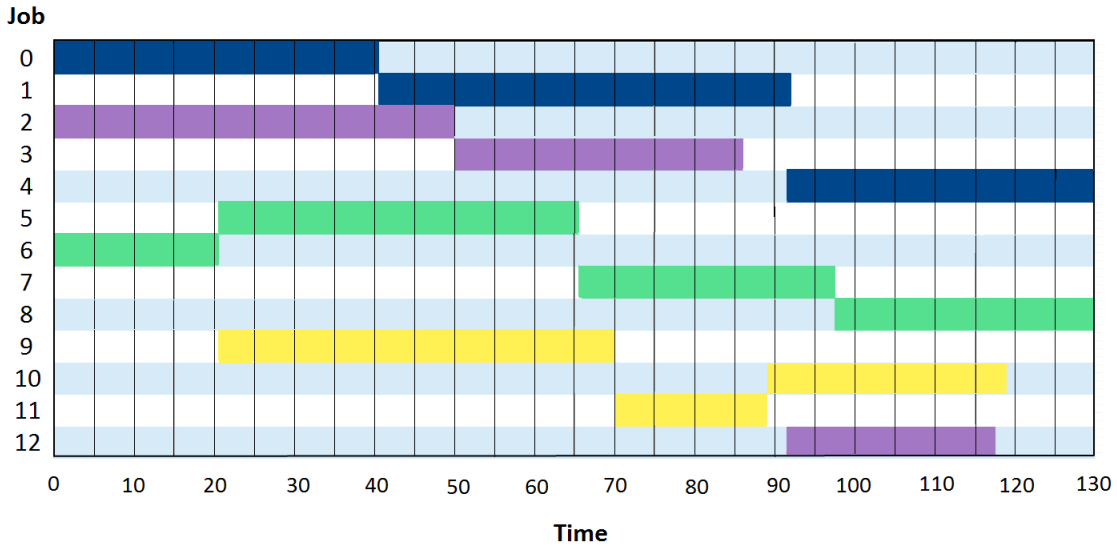

Figure 3.2: Gantt chart which, after inspecting Figure 3.1, minimises the number of production lines required to complete all jobs

For convenience, I have included the start and finish times which give this optimal workflow in the table below, whereby optimal workflow means we have minimised the total time required to complete all 13 jobs, and within that minimised the number of production lines required so that we can execute as many jobs as possible in parallel to minimise time wastage in terms of stationary production lines.

Job	Optimal start time	Optimal finish time
0	0	41
1	41	92
2	0	50
3	50	86
4	92	130
5	21	66
6	0	21
7	66	98
8	98	130
9	21	70
10	89	119
11	70	89
12	92	118

Also note that these optimal times and workflow in terms of distributing the jobs across production lines are not entirely unique; for example, job 12 could start at time 100 and still finish before time 130, or job 8 could be completed by the purple production line in Figure 3.2 at start time 92 and job 12 by the green production line in Figure 3.2 at start time 98. In both cases, all jobs are completed by time 130 using four production lines, highlighting the fact that the optimal workflow is not unique.