

Scientific Computing (M3SC) Project 2

Omar Haque

March 23, 2017

1 INTRODUCTION

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 sections and describing the code and theory simultaneously.

2 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 2.1

```

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

```

3 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 $m \in \{0, 1, \dots, 12\}$ I write m_s and m_f for the start and finish node for job m respectively. I also write v_s and v_f for the virtual start and virtual finish node respectively.

Figure 2.1: np.array produced from extract_data

```
[[0 41 [1, 7, 10]]
 [1 51 [4, 12]]
 [2 50 [3]]
 [3 36 []]
 [4 38 []]
 [5 45 [7]]
 [6 21 [5, 9]]
 [7 32 []]
 [8 32 []]
 [9 49 [11]]
 [10 30 [12]]
 [11 19 []]
 [12 26 []]]
```

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
9     """
10    global total_nodes, virtual_start, virtual_finish, job_duration
```

```

11 # we start with an array of -1s and populate the entries that
12 # correspond to connected nodes.
13 total_nodes = data.shape[0]
14 weight_matrix = -1 * np.ones((2 * total_nodes + 2, 2 * total_nodes + 2)
15                               , dtype=int)
16
17 # node start is the index of start jobs
18 node_start = data[:, 0].astype(int)
19 # job_duration are the job durations for each job
20 job_duration = data[:, 1].astype(int)
21
22 # joining each node start to node finish, with the weight as that job's
23 # duration.
24 weight_matrix[node_start, node_start + 13] = job_duration
25
26 # note on efficiency: I could perhaps do the following for
27 # loop by flattening the data[:,2] column, but I need to
28 # create a list of node_start corresponding to the number of
29 # jobs that each job depends on. This is O(N) anyway, so doing
30 # this via a for loop isn't slower.
31
32 # note: also, python doesn't like slicing like a[0,[[4,5],[1,2,3]]]
33
34 for row in data:
35     jobs2 = row[2]
36     node_start2 = row[0]
37     for job in jobs2:
38         # this is the connections of dependent jobs
39         weight_matrix[node_start2 + 13, job] = 0
40
41 # these are the indices for virtual start and finish
42 virtual_start = int(weight_matrix.shape[0]) - 2
43 virtual_finish = int(weight_matrix.shape[0]) - 1
44
45 # allow movement between virtual start and all the nodes
46 weight_matrix[virtual_start, 0:total_nodes] = 0
47 # allow movement from all the nodes to virtual finish
48 weight_matrix[total_nodes:virtual_start, virtual_finish] = 0
49
50 return weight_matrix

```

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

```

1 weights = generate_weight_matrix(data)

```

4 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 3, 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 in Figure 4.1:

Figure 4.1: 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 .

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

printing this longest path in Figure 4 gives [0,1,4].

Figure 4.2: 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. Namely, $[v_s, node1_s, node1_f, node2_s, node2_f, \dots, nodek_s, nodek_f, v_f]$. 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 $0 \rightarrow 1 \rightarrow 4$ is the longest string of jobs.

5 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

$$b_m = m_s \rightarrow m_f$$

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

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

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

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

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

Proof: Suppose not. Then there exists a longer sequence to job i : $b_{k_1} \rightarrow \dots \rightarrow b_{k_p} \rightarrow b_{m_i}$, say. But then, $b_{k_1} \rightarrow \dots \rightarrow b_{k_p} \rightarrow b_{m_i} \rightarrow b_{m_{i+1}} \dots \rightarrow b_{m_{l-1}} \rightarrow b_m$ 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.

Due to these results, after finding the longest job sequence in the graph [0,1,4], we can simply remove the edge from $4_f \rightarrow v_f$. Then if we find the longest job sequence in the graph again, there will be no route from v_s to v_f via job 4, and we will find the second longest job sequence in the graph. Continuing in this way, removing one node at a time we can find all of the necessary remaining (shorter) job sequences to recover the earliest start and stop times.

In fact, we can remove more connections than just each job in the job sequence to the virtual finish node. After we have found the longest job sequence finishing at job e , $b_c \rightarrow b_d \rightarrow b_e$, say. We can remove the connection between the last pair of jobs in this path, d and e . For if there were another job sequence which could utilise this edge, $b_c \rightarrow b_d \rightarrow b_e$ would not be the longest job sequence in the graph (currently). This reduction means that there is one less edge for the Bellman Ford algorithm to iterate through, and slightly eases the time complexity.

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.
2. By the arguments above, you have the start and stop times for every job in that path.
for (node in longest path) { update the earliest start time for that node if you haven't already }
3. for (node in longest path) { remove connection from node to virtual finish. IF len(longest path) > 1: remove the edge between the last pair in the longest path. }
4. If there are still jobs whose start time you haven't determined, GOTO 1. Else, finish.



It occurred to me that another possible optimisation in this algorithm would be to iterate backwards through the longest job sequence, as you could terminate this loop as soon as you have reached a job whose time you have already settled. But this would not improve the time complexity in this situation for a number of reasons: In order to attack the problem this way you need to calculate the sum of the longest job sequence, which is an order of n operation anyway. Sure, `np.sum` is fast but the job sequence is not a numpy array, and converting each sequence to an array would cost even more time. This approach may be more effective when job sequences and dependencies are very large and complicated, but in this situation the overhead is too much for it to be useful.

Here is the python implementation for this section.

```
1 def iterative_bell(adjusted_weights, start_stop):  
2     """
```

```

3      This function uses the Bellman Ford algorithm to iteratively find
4      the remaining job sequences in the graph. It adjusts the start_stop
5      array
6
7      :param adjusted_weights: A, the negative adjacency matrix
8      :param start_stop: the list of earliest start and stop times to be
9      edited and updated
10     :return: the longest paths used to find the start_stop times
11     """
12
13     # create the longest paths list
14     longest_paths = []
15
16     # create a copy of the weight matrix
17     temp_weights = np.copy(adjusted_weights)
18
19     # create a boolean array of False's. removed_nodes[m] = True if
20     # job m has been removed (its value of earliest start and stop
21     # has been calculated).
22     removed_nodes = np.zeros(13, dtype=bool)
23
24     # a counter so we know when to stop.
25     counter = 0
26
27     while counter < 13: #
28
29         # find the longest path in the graph from virtual start
30         # to virtual finish
31         job_sequence = updated_bellman_ford(virtual_start,
32                                           virtual_finish,
33                                           temp_weights)[1:-1:2]
34
35         # remove the connections from those jobs to virtual finish
36         temp_weights[np.array(job_sequence)+13,virtual_finish] = 1
37         # if we have at least 2 elements, remove the last pair
38         if len(job_sequence) > 1:
39             temp_weights[job_sequence[-2] + 13,job_sequence[-1]] = 1
40
41         # determine the times using the algorithm derived
42         current_time = 0
43
44         # iterate through the jobs in the job sequence
45         for job in job_sequence:
46

```



```

47         # only add to start_times if you haven't already
48         if not removed_nodes[job]:
49             # set it to the current time. i.e the sum of jobs before it
50             start_stop[job, 0] = current_time
51             start_stop[job, 1] = current_time + job_duration[job]
52             # add 1 to the counter
53             counter += 1
54
55         # update current_time
56         current_time += job_duration[job]
57
58         # set all of the nodes in this job sequence to True
59         removed_nodes[job_sequence] = True
60         # add this job sequence to the list of longest paths
61         longest_paths.append(job_sequence)
62
63     return longest_paths

```

We store the longest paths to help with the creation of the Gantt diagram later.
This is the code to run this function

```

1     start_stop = np.zeros((13, 2), dtype=int)
2
3     longest_paths_list = iterative_bell(adjusted_weights, start_stop)
4
5     full = np.column_stack((data, start_stop))

```

See Figure 5.1 for the start_stop array.

Figure 5.1: earliest start and stop times for each job - row with index i corresponds to job i

```

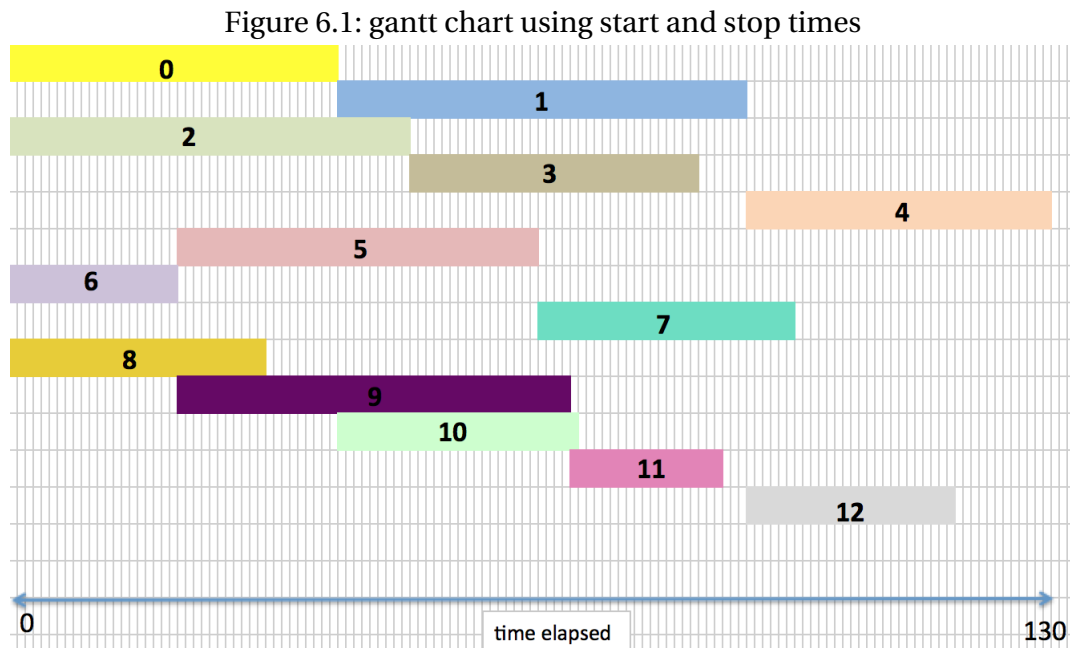
[[ 0 41]
 [41 92]
 [ 0 50]
 [50 86]
 [92 130]
 [21 66]
 [ 0 21]
 [66 98]
 [ 0 32]
 [21 70]
 [41 71]
 [70 89]
 [92 118]]

```

6 PRODUCING THE GANTT CHART

Now we have our earliest start and stop times for each job, we can find one solution using a worker for each job, this finishes in the fastest time of 130 minutes. We are bound to 130 minutes because this is the earliest finishing time for job 4. See Figure 6.1.

We have optimised for time, but clearly not for workers.



At the end of section 1.4 we showed how the *iterative_bell* function found all of the longest job sequences in our graph. I print these in Figure 6.2

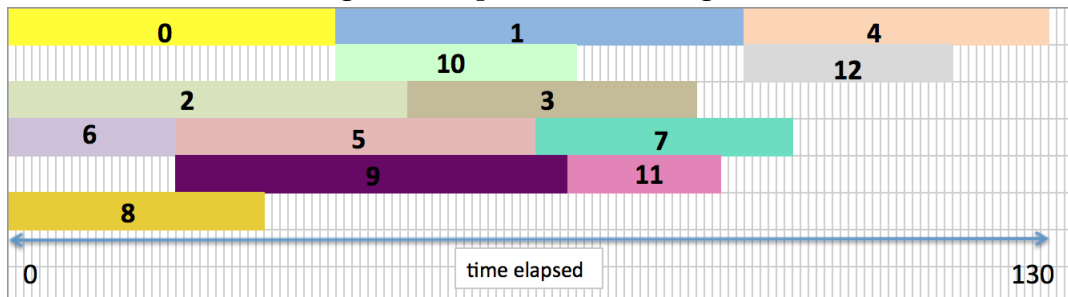
Figure 6.2: The longest job sequences in the graph

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

Now, we can slide jobs up and down in the Gantt diagram as we please, without disturbing times or job dependencies. We should do so in accordance with the longest job sequences, in figure 6.2 as we will inevitably waste less time between jobs. This sliding up and down results in the Gantt diagram in figure 6.3

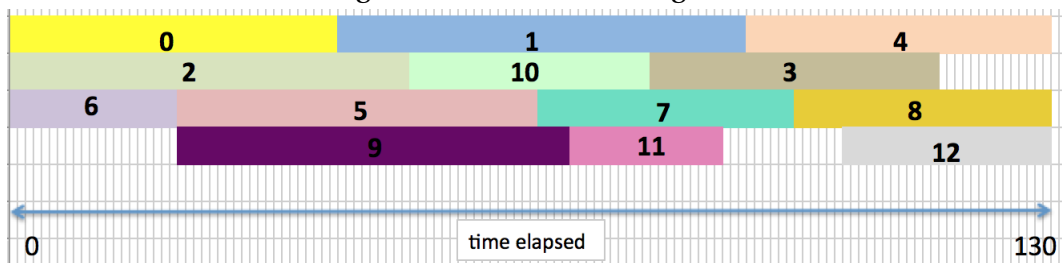
Some jobs are able to move freely as no jobs depend on them, and they depend on no others. Like job 8 for example. So we can move job 8 to the end of the $6 \rightarrow 5 \rightarrow 7$ row as

Figure 6.3: updated Gantt diagram



its duration will still be less than 130. We then spot that we can move 12 to the end of the $9 \rightarrow 11$ row, and slot 10 between $2 \rightarrow 3$ while still respecting all job dependencies. This is shown in figure 6.4.

Figure 6.4: Final Gantt diagram



It is impossible to fit these jobs on three lines even if we ignore job dependencies, so we have optimised the workflow as required.