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
```

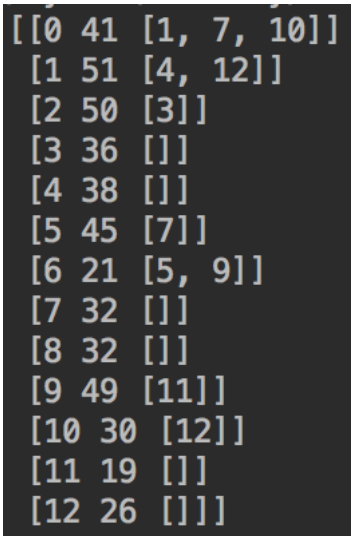
Running this code then produces the array shown in figure 1.1

```
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.

```
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 and returns the longest paths used.
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     # This is a list of 13 Falses
20     # If a node appears in a job sequence (i.e. we know its start time)
21     # it turns to True.
22     removed_nodes = np.zeros(13, dtype=bool)
23     # 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)
        longest_paths.append(job_sequence)
        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

```
    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

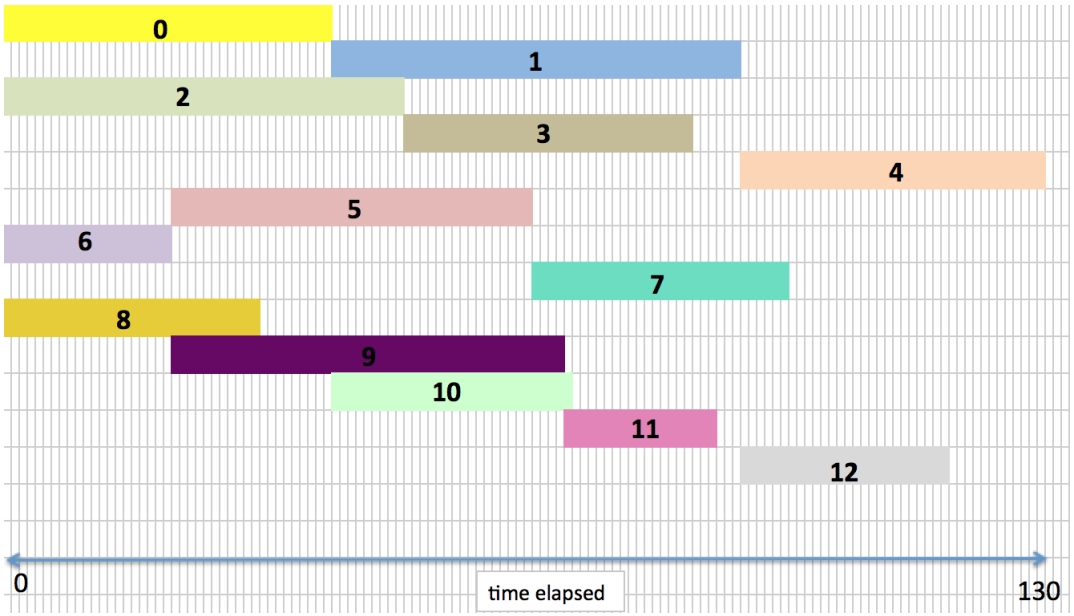
Figure 1.4: earliest start and stop times for each job

```
[[  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

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.
We have optimised for time, but clearly not for workers. We do this now.

Figure 1.5: gantt chart using start and stop times



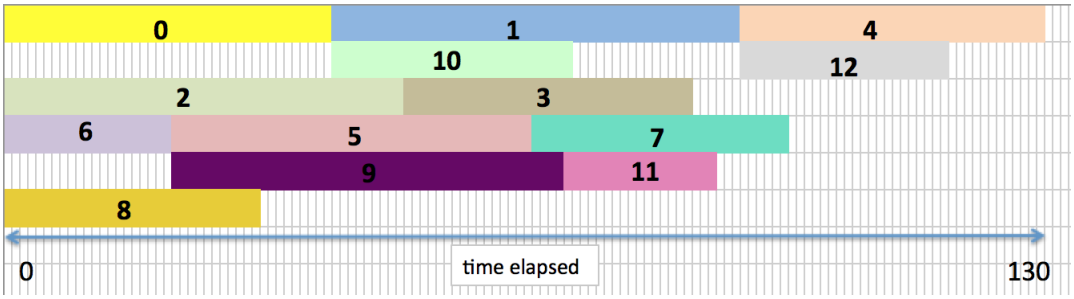
At the end of 1.4 we showed how the *iterative_bell* function found all of the longest job sequences in our graph. I print these here:

Figure 1.6: The longest job sequences in the graph

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

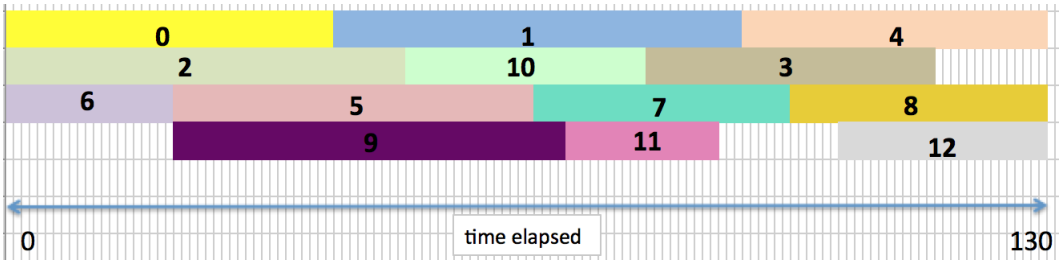
Now, we can slide jobs up and down in the Gantt diagram as we please, without disturbing times or job dependencies. We should do so in accordance with the longest job sequences, in figure 1.6 as we will inevitably waste less time between jobs. This sliding up and down results in the following Gantt diagram.

Figure 1.7: updated Gantt diagram



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 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.

Figure 1.8: 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.