

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 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 $m \in \{0, 1, \dots, 12\}$ I write m_s and m_f to mean the start and finish node

Figure 1.1: 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 []]]
```

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
```

```

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)
```

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 .

```
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 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 $0 \rightarrow 1 \rightarrow 4$ 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

$$b_m = m_s \rightarrow m_f$$

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.

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):
2
3     # create a copy of the weight matrix
4     temp_weights = np.copy(adjusted_weights)
5
6     # This is a list of 13 Falses
7     # If a node appears in a job sequence (i.e. we know its start time)
8     # it turns to True.
9     removed_nodes = np.zeros(13, dtype=bool)
10    # a counter so we know when to stop.
11    counter = 0
12
13    while counter < 13: # O(1)
14        # find the longest path in the graph from virtual start
15        # to virtual finish
16        job_sequence = updated_bellman_ford(virtual_start,
17                                           virtual_finish,
18                                           temp_weights)[1:-1:2]
19        # O(13*E) # all we can change is E
20
21
22        # remove the connections from those jobs to virtual finish
23        temp_weights[np.array(job_sequence)+13,virtual_finish] = 1
```

```

24
25     # determine the times using the formula derived
26     current_time = 0 # 0(1)
27
28     # iterate through the jobs in the job sequence
29     for job in job_sequence: # 0(k)
30
31         # only add to start_times if you haven't already
32         if not removed_nodes[job]: # 0(1)
33             # set it to the current time. i.e the sum of jobs before it
34             start_stop[job, 0] = current_time # 0(1)
35             start_stop[job, 1] = current_time + job_duration[job]
36             # add 1 to the counter
37             counter += 1 # 0(1)
38
39         # update current_time
40         current_time += job_duration[job] # 0(1)
41
42     # so the for loop is 0(k) in total
43
44     removed_nodes[job_sequence] = True # 0(1)

```

This is the code to run this function

```

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