IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

M3SC Project w: – Traffic Flow Through A Simplified Street Network

Christopher McLeod

March 23, 2017

Abstract

This project seeks to find a solution to a scheduling optimization problem. In this scenario, there are a number of tasks for multiple production lines. These tasks have given durations and dependencies, e.g., tasks 4 and 12 depend on the output of task 1, and thus in the work-flow task 1 has to come before tasks 4, and 12. With this in mind, an attempt is made to execute as many tasks in parallel, to save time. Simultaneously, an effort is made to do this in the most efficient way, i.e. minimizing the number of 'workers' it would take to implement the optimized system.

CONTENTS

1 BUILDING THE SYSTEM

In this part, following the instructions I attempt to create and explain the reasoning for setting up a network which will assist in optimizing the work-flow for the parallelized system.

1.1 JOBS AS START AND FINISH NODES

The first step in optimizing the problem is seeing the problem as a directed graph. This means seeing jobs as a start and and end node with a weight connecting them, visualized below [1]

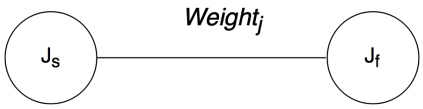

Figure 1.1: A job represented as a start and finish node

j_s is the node representing the start of the job, likewise j_f represents the end of the job. Above the arc connecting the start to the end of the job is the *'weight'* of the job, $weight_j$ i.e. its duration. Note this the directional arrow is implied here. It is not shown explicitly for the sake of visibility. Then doing this for all jobs, we generate the skeleton of a network shown in Figure 1.2.

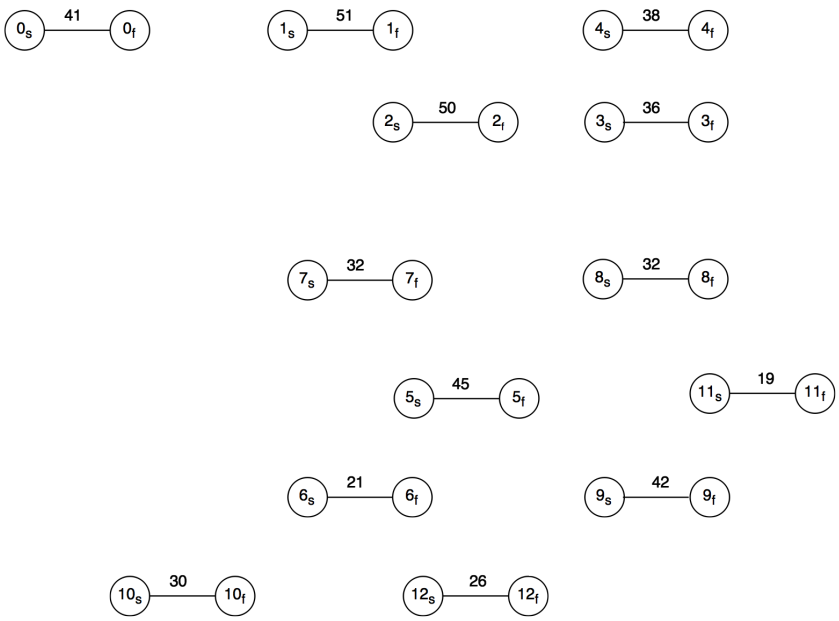

Figure 1.2: Initial system of start and end nodes

[1] All diagrams were produced with http://draw.io, a free online diagram software for making flowcharts

1.2 Connecting the Network

Now the useful concept of viewing jobs as weighted edges from start to end has been introduced, it can be taken further. I connect i_f to j_s (shorthand $i_f \rightarrow j_s$) *iff* job j depends on the completion of job i. Below is the network generated by the dependencies of the jobs in this way [2]

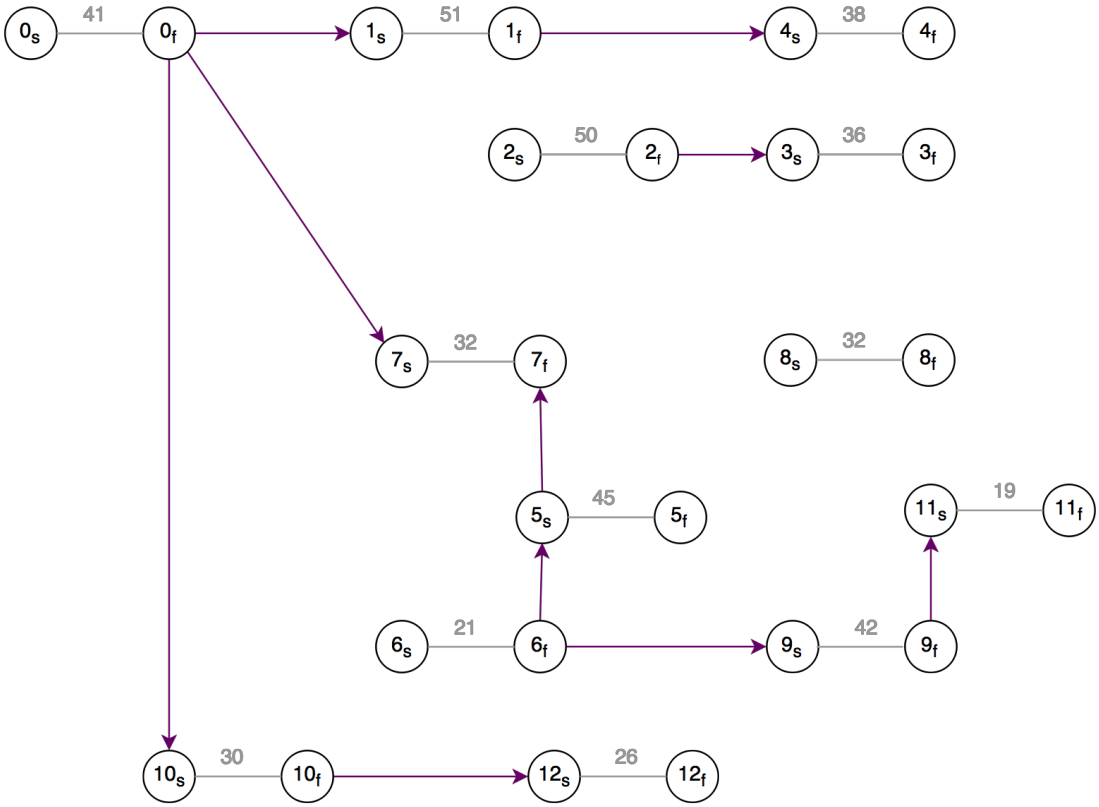

Figure 1.3: The directed job network

This way, I generate a network in which one only gets the chance to start a job if its prerequisite jobs have been completed. This is clear if one thinks about the problem recursively. For example. Assuming I start at a job with no prerequisites, say job 0, then the only way I can end up starting job 12 is if the prerequisite jobs 0 and 10 have been completed.
Start nodes of valency 1 are the ones with no dependency; this means we can not easily find a path through the network which gives the longest length in the graph. Analogously, I can not use this to find the shortest time of the parallel processes by any definite means.

[2] Note the value of the arcs $i_f \rightarrow j_s$ is zero, i.e i is connected to j with no weight (it takes no time to start the next job), so an implicit weight of 0 is omitted from the diagram.

1.3 Connecting all Jobs Virtually

To resolve the problem, I introduce a 'virtual' start and finish node. The virtual start node arcs to all 'start'-nodes and the virtual end node is arced to by all 'finish'-nodes of a job

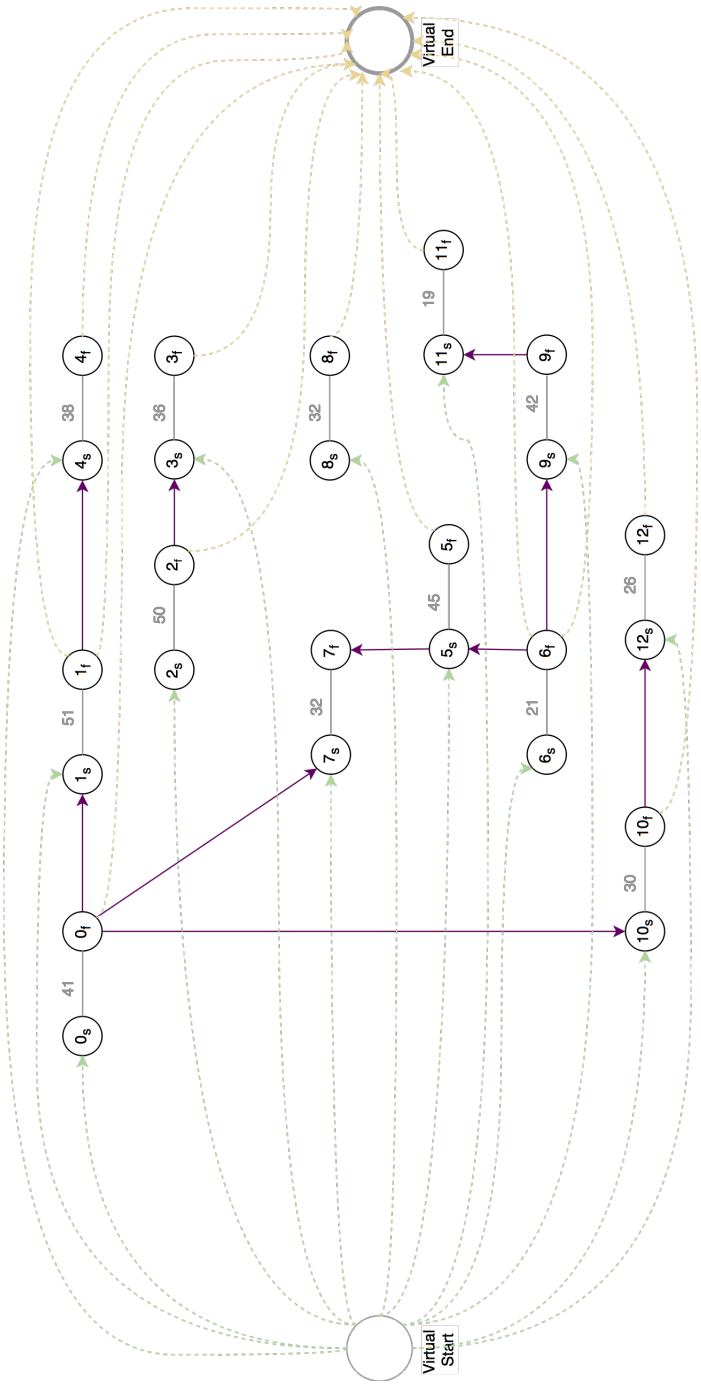

Figure 1.4: The fully connected (virtual) system

Now there is a point to begin and end any traversal through the network. Analogously, I can

now find the lengths of time it takes to do any valid chain of jobs. By valid, I mean that if $k_s \to k_f$ is traversed (job k is done) then all arcs feeding into the chain of prerequisite nodes has also been traversed (i.e. all of job k's prerequisite jobs have also been done). The Plan from here is to find the paths in decreasing length from the virtual start to the virtual end. The longest one gives me the longest string of jobs, ensuring I can best parallelize the code in accordance with this path.

1.4 Coding the System

To find the paths through the network in decreasing length, the plan is to use the Bellman-Ford algorithm on the *negative* adjacency matrix of the above virtual system. This is because the minimum values become the largest and vice versa. First, I create the positive adjacency matrix as follows

```
1  import numpy as np
2  import sys
3
4  # create nominal '0' for connectivity without weight
5  zer0 = sys.float_info.min
6
7  # create an ordered list of durations
8  duration = [41,51,50,36,38,45,21,32,32,49,30,19,26]
9
10 # initialise the connectivity/weight matrix for the start
11 # /finish job system
12 adj = np.zeros((28,28))
13
14 # initialise the off-kilter diagonal for the s/f connections
15 # [1,2],[3,4],...
16 amm = np.zeros(27)
17 # load the values into this dummy vector
18 amm[1::2] = duration
19 # insert this vector into the diagonal above the main
20 adj = np.diag(amm,1)
21
22 # connect the end of all jobs to the virtual end node.
23 # j a job => start nodes have form 2j+1, ends have form 2j
24 # except for j=0
25 # adj[0,:] the virtual start arcs, adj[:,27] the virtual end
26 # arcs
27 adj[0,1::2] = zer0
28 adj[::2,27] = zer0
29
30 # vectorise the componentwise arcs given by order dependency
```

```
conx = [2,2,2,4,4,6,12,14,14,20,22]
cony = [3,15,21,9,25,7,15,11,19,23,25]

# assign nominal connectivity value for these dependencies
adj[conx,cony] = zer0
```

The system minimum float is introduced as a nominal value which signifies two nodes are connected by an arc of weight 0. I used this value because even when I sum later, this value does not show up: the display rounds well before it gets the chance to display the 308^{th} decimal place ($sys.float_info.min = 2.2250738585072014e-308$).
For the connections $i_s \rightarrow i_f$, the weight of the connection is input from a vector called 'amm'. In essence, the adj matrix has its 0 and 27 sliced out, the diagonal above the main is then filled with a 0 if the column entry is even and duration[c] if c is the column entry which is also odd. The rest is straight forward from the comments in the code, it is merely simple vectorization and the knowledge that for j in jobs creating j_s and j_f and placing them adjacent column-wise is the same as mapping j to $2j+1$.
Next, simply take a negative copy of this and we have an operational weight/connectivity matrix for the Bellman-Ford algorithm to work

```
import BellmanFord as bf
import copy as cp

# make the adj matrix negative (for Bellman-Ford)
Wei = -cp.copy(adj)
```

The reason Bellman-Ford is necessary is because as a greedy algorithm, Dijkestra doesn't have the ability to deal with the negative weight matrix appropriately, whereas Bellman-Ford is a relaxation algorithm and considers all possibilities subsubsectionDigression: BF Vs Dijk Dijkstra is a faster algorithm than Bellman-Ford, so I feel I need to justify that the use of it is appropriate. The main justification is that the network is small in this case. The following test code

```
from time import time as t
import BellmanFord as bf
import Dijkstra as di
import copy
import numpy as np
import sys

# create adj matrix in the normal way (pathFinder.py)
zer0 = sys.float_info.min

duration = [41,51,50,36,38,45,21,32,32,49,30,19,26]
adj = np.zeros((28,28))
amm = np.zeros(27)
amm[1::2] = duration
```

```
15 adj = np.diag(amm,1)
16 adj[0,1::2] = zer0
17 adj[::2,27] = zer0
18
19 conx = [2,2,2,4,4,6,12,14,14,20,22]
20 cony = [3,15,21,9,25,7,15,11,19,23,25]
21
22 adj[conx,cony] = zer0
23
24 Wei = - copy.copy(adj)
25
26
27 # use system time to time Dijkstra's algorithm on pos.
28 # conn/wei matrix
29 t00 = t()
30 print 'Dijk path;', di.Dijkst(0,27,adj)
31 t01 = t()
32
33 # use system time to time the Bellman-Ford algorithm on neg.
34 # conn/wei matrix
35 t10 = t()
36 print 'Bell path:', bf.BellmanFord(0,27,Wei)
37 t11 = t()
38
39 # average the time by path length
40 t0 = (t01-t00)/len(di.Dijkst(0,27,adj))
41 t1 = (t11-t10)/len(di.Dijkst(0,27,Wei))
42
43
44
45 if __name__ == '__main__' :
46     print 'Dijkstra Takes', t0,'s'
47     print 'Bellman Ford Takes', t1,'s'
48     print 'Dijkstra is', (t1/t0),'times quicker'
```

When run a handful of times gives a speed-up (relative to the path-length) typically produces results such as

```
Dijkstra Takes 0.000690579414368 s
Bellman Ford Takes 0.00125387310982 s
Dijkstra is 1.81568272052 times quicker
```

To whit, the speed up is less than a microsecond of for this system.
End digression.

To put this matrix to use, I write the code

2 THE LONGEST PATH

```
# run through the jobs
for i in range(13):

    # find the shortest path on the neg. conn/wei matrix with
    # Bellman-Ford
    shpath = bf.BellmanFord(0,27,Wei)
    # delete the connection of the last done job to the
    # virtual finish node
    penult = shpath[-2]
    Wei[penult,27] = 0
    SHPATH.append(shpath)

    # one uses a filter to regain the jobs from the 'double'
    # node system
    one = (np.array(filter(lambda x: x%2==1 and x<27,shpath)) - 1)/2
    # two uses a list comp. to list the durations
    two = [duration[x] for x in one]
    # pathTime sums the durations/weights traversed, giving
    # path length
    pathTime = sum(two)
    print pathTime
```

I simply run Bellman-Ford ('BF') for the system and sequentially, at the next longest path, delete the connection to the virtual end node. This forces the BF algorithm to take the *next* longest path in the system, and thus a sequence of paths of decreasing length is produced. To illustrate this clearly, I created 'one' and 'two' and 'pathTime' which respectively find the jobs, create a list comprehension of their durations in the path and then sum that list to give the time taken to do the whole list. This gives the following

```
path: [0, 1, 2, 3, 4, 9, 10, 27]
path time: 130
path: [0, 1, 2, 3, 4, 25, 26, 27]
path time: 118
path: [0, 13, 14, 11, 12, 15, 16, 27]
path time: 98
path: [0, 1, 2, 3, 4, 27]
path time: 92
```

```
path: [0, 13, 14, 19, 20, 23, 24, 27]
path time: 89
path: [0, 5, 6, 7, 8, 27]
path time: 86
path: [0, 1, 2, 21, 22, 27]
path time: 71
path: [0, 13, 14, 19, 20, 27]
path time: 70
path: [0, 13, 14, 11, 12, 27]
path time: 66
path: [0, 5, 6, 27]
path time: 50
path: [0, 1, 2, 27]
path time: 41
path: [0, 17, 18, 27]
path time: 32
path: [0, 13, 14, 27]
path time: 21
```

So I have produced the path giving optimal parallelization of the program. From here I have also iteratively determined the remaining (shorter) job sequences, until I determined the start and finish times of all jobs.
From here I can produce the array of optimal start and finish times for each job. Outside the job loop, initialize

```
# initialize T, the optimal array detailing Job|Start|End
T = np.zeros((13,3))

# create a list of 'unseen' nodes - this is all of them before
# anything happens
unseen = np.arange(13)

# assign values of the jobs
T[:,0] = unseen
```

The format of T is to be Job| optimal Start| optimal End. This is achieved inside the node loop by

```
for i in range(13):
        # fill the optimal start/end times for jobs in T
    # cut shpath to give the jobs done
    for n in shpath[1:-1:2]:
        # check this job has not been done already
        if (n-1)/2 in unseen:
            # if it's not the first job in the path
            if shpath[1:-1:2].index(n)>0:
```

```
                # assign its start value to the duration of
                # previous jobs in the path
                T[(n-1)/2,1] = sum(two[:shpath[1:-1:2].index(n)])

            # if it's the first job in the path
            else:
                #the optimatal start time must be zero
                T[(n-1)/2,1] = 0

        # the optimal end time is nothing more than the
        # optimal start plus the duration of the job being
        # undertaken
        T[(n-1)/2,2] = T[(n-1)/2,1] + duration[(n-1)/2]

        # make this node 'seen' so it will not be evaluated
        # again
        np.delete(unseen,(n-1)/2)
        ...
print '   Job |Start| End'
print T
```

The code fills The optimal start entry for each job by checking if it is the first job in the path (at which point it's optimal start is at 0) or, if not, sums the durations of the jobs before it. The optimal end is then calculated by simply adding on the duration of that job. From this the array generated is

```
   Job |Start| End
[[   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.]]
```

T is the most useful piece of information so far. It tells one exactly when the absolute earliest a job can start. A rough visualization via the code

```
import matplotlib,pyplot as plt
# Immediate Workflow diagram
# plot for jobs in order the optimal start and end against time
plt.hlines(T[:,0],T[:,1],T[:,2],linewidth=30, color='red')
plt.xticks([0]+list(T[:,2]))
plt.yticks(T[:,0])
plt.grid()
plt.ylim([min(T[:,0])-1,max(T[:,0])+1])
plt.show()
```

yields the plot Figure 3.1. This rough plot gives a feel for what our optimized/parallelized

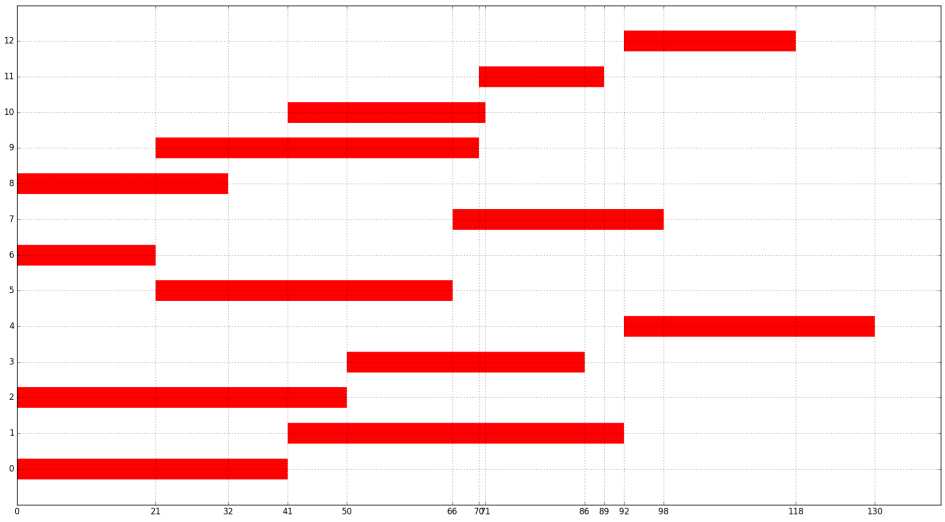

Figure 3.1: non-Compressed Gantt Chart

workflow might look like, if one has a mind for fitting shapes in ones head. The next task is, using the optimum stop and start matrix, to compress this into its most parallelized state, with the jobs all culminating at time is 130.

3.1 COMPRESSING THE WORKFLOW

The first thing that must be done is assign all jobs with no pre-requisite jobs to new 'workers' (i.e. the people/units/companies actually completing the jobs). From here, at any given point, one wishes to find out which worker will finish soonest and assign them the

next job first. Doing this repetitively will give the parallelized workflow with the least amount of wasted time, i.e. fewest workers and rests that the workers take. That is, of course, given that they are assigned the right jobs!
The methodology of the below code is to, at any time step, pick the worker that is going to finish soonest. From here there are 3 types of jobs to assign them, which have 1 quality (duration). The three types of jobs are: those that can optimally be undertaken at the end of the worker's current job, those whose optimal start times are before the worker finishes their current job, and those whose optimal start time is after.
Preferably, I want to minimize gaps, and so the most preferable next job to assign a worker is one that can optimally start as soon as they finish their current job. Next, the one with the closest optimal starting time before the finish of the current job is most favorable, because I do not want to leave this job sitting and stop other tasks from being completed which upon it rely. The least favorable job is the the one for which the worker has to wait to start, so the one with least waiting time is assigned.
If there is more than one job, in the case that there is a wait until the next job, we wish to pick the shortest duration job, that we may get a more favorable job more quickly in the future. In the other two cases however, we take the one of longest duration to ensure as few gaps as possible and make the last case less likely for another worker.
The code below encapsulates this methodology. With the above in mind and the comments below, I am hoping the reader finds the algorithm explained.

```
# initialize the work-flow for each worker (nested list)
workers = []

# assign jobs and optimally start/end normally from T
jobs=T[:,0]; start = np.array(T[:,1]); finish = np.array(T[:,2])

# make 'todo' the list of jobs left to do
todo = map(int, jobs)
# initialize a list for the cumulative time each worker has
# spent
cumtime=[]

# assign the jobs with no prerequisites to new workers
for j in np.where(start==0)[0]:
    workers.append([j])
    cumtime.append(duration[j])

# these jobs have been done, so remove them from todo
todo=np.delete(todo,np.where(start==0)[0])

# if there is a wait before a worker can do their next job,
# restloc will
```

```
23 # record for which job the 'rest' occured and for how long
24 restloc=np.zeros(13)
25
26 # flag is true as there are jobs to do
27 flag=True
28
29 while flag is True:
30     # cumloc looks for the worker that will finish first
31     # using argmin
32     cumloc = np.argmin(cumtime)
33     # w is the worker to next to do something as they will
34     #finish first
35     w = workers[cumloc]
36     # find the difference in the start times of the jobs left
37     # to do
38     # and the ending of the 'current' job the worker's doing
39     diff = start[list(todo)] - finish[w[-1]]
40
41     # find where (if anywhere) the difference is zero
42     zeroWhere = list(np.where(diff==0)[0]);
43     zeroWhere = map(int, zeroWhere)
44     if not set(zeroWhere):
45         # failing this, look for where the diff is negative
46         negWhere = np.where(diff<0)[0]
47         # create the 'class' of these negatives
48         negClass = diff[negWhere]
49         # for convinience, make them integer values
50         negClass = map(int, negClass)
51         if not set(negWhere):
52             # do same for pos diff
53             posWhere = np.where(diff>0)[0]
54             posClass = diff[posWhere]
55             posClass = map(int, posClass)
56
57     # test in order of preference
58     if set(zeroWhere):
59         # of this group, find the job that takes the longest
60         nextjob = list(duration).index(max([duration[t] for t
61                                             in todo[zeroWhere]]))
62         # diff = 0, no rest necessary
63         rest = 0
64
65     elif set(negWhere):
66         # shortlist the location in the class of jobs with
```

```
67          # least neg. diffs
68          shortlist = np.where(np.array(negClass)==max(negClass))[0]
69          # find the job in todo with the greatest duration
70          nextjob = todo[list(negWhere[list(shortlist)])][0]
71          # optimum time is less, so no need to rest before doing
72          # it
73          rest = 0
74      elif set(posWhere):
75          # shortlist the location in the class of jobs with least
76          # pos. diffs
77          shortlist = np.where(np.array(posClass)==min(posClass))[0]
78          # find the job which has minimum duration
79          nextjob = todo[list(posWhere[list(shortlist)])][-1]
80          # diff positive so must wait until it becomes possible
81          # to do the job
82          rest = start[nextjob]-finish[w[-1]]
83
84      # assign the rest to the locator at the position of the
85      # next node (pre-emptive)
86      if rest>0:
87          restloc[nextjob] = rest
88
89      # remove from todo the job that's now done/planned
90      todo=np.delete(todo,np.where(todo==nextjob)[0])
91      # give the worker the 'next job'
92      w.append(nextjob)
93
94      # update cumtime
95      cumtime[cumloc] += duration[nextjob] + rest
96
97      # check we still have jobs to do, if not,
98      #leave the process: done
99      if not set(todo):
100         flag = False
101         break
102
103 print 'workers:', workers
```

The result of which is

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

And so I have a parallelized scheme of workers. This can now be represented as a compressed Gantt graph, a good visual workflow.

The following fairly mundane code does the job of plotting it

```
# plot the compressed gantt chart
# pick colors for the job blocks
colorcycle=['cyan','magenta']

plt.figure()

# assign dummy variables and initialize 'ends'
ends=np.arange(14); m = 1; j=1

#run through the workers
for i in range(len(workers)):
    #reset the cumulative durations
    addon = 0

    #plot block for each worker
    for w in workers[i]:

        # re-order workers
        offset = len(workers)-i
        # plot the worker's jobs. start is sum of the duration
        # of previous jobs and
        # potential rest times. end time is the same plus the
        # duration of the job
        plt.hlines(offset, addon+restloc[w],
                                restloc[w]+addon+duration[w],
                                        colors=colorcycle[m] ,lw=25)

        # simply label blocks
        plt.annotate('job %i' %w,(addon+restloc[w]+0.5*duration[w]-4,
                                                        offset-0.03))
        # record the end of the block for future visualization
        ends[j]=restloc[w]+addon+duration[w]
        addon += duration[w]
        # change color of next block
        m=(m+1)%2
        j+=1


plt.ylim(-0.5,len(workers)+0.5)
plt.ylabel=workers
plt.grid(which='both')
```

```
42 plt.xticks(ends)
43
44 plt.show()
```

In brief, it runs through the workers, calculates the start time of a worker's job as the sum of the durations of previous jobs and potential rest times it may have needed. The end time is then the same plus the duration of the job being carried out. A neat bit of modular arithmetic switches colors from cyan to magenta for the blocks. 'ends' (which records the end time of any job) is also useful as it adds to the visualization of the graph, giving specific start and end time markers on the x axis for the jobs.
Finally, the long awaited Gantt graph is ready to be displayed!
So, Figure 3.2, the long awaited Gantt chart has been created. It seems optimal as very little time is wasted. We also know the algorithm optimized the parallelization in terms of number of workers as well, as

```
In [801]: sum(duration)/130.0
Out[801]: 3.6153846153846154
```

Rounding this up to 4, the minimum number of workers for that time flow is 4!
And thus, the problem is sovled.

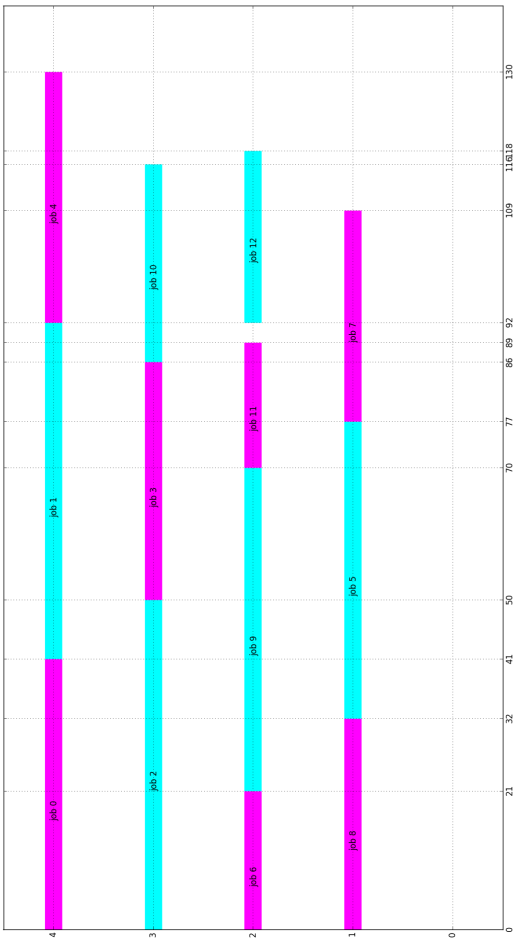

Figure 3.2: The compressed Gantt Graph