Rahul Jaisimha

rj7994 **Study Group: Simon Kumets**

EE360C – Dr. Ghosh **Sarang Bhadsavle**

Nishanth Shanmugham

Final Project

PROBLEM 1:

i) For the following processes, the greedy solution is 5 but the optimal solution is 4(by putting the process with d = 3 before the process with d = 1.

t = 4; d = 1	t = 1; d = 3

ii) Use the recurrence equation:

$$minLateness(S) = \begin{cases} \min\left(minLateness(S-e) + (e.C - e.d), \ldots\right), if\ e\ were\ at\ the\ end \\ \qquad\qquad for\ all\ elements\ e\ in\ Subset\ S \\ 0, \qquad\qquad if\ set\ S\ is\ empty \end{cases}$$

where S is a subset of processes of arbitrary size. (e.C is e's completion time and e.d is its deadline. Set S is the set S with element e removed)

To use DP, we solve bottom-up. For S of size 1, we have n different sets. For all S of size 2, we can use the smaller subsets to compute the minLateness for that (unordered) set of processes, using the recurrence equation.

```
null set <- 0
for i from 1 to N
   for every subset S  of processes of size i
      S.minLateness <- min((all subsets of S of size
               i-1).minLateness + lateness of i^th element
backtrack
```

This algorithm runs in **$O(2^n)$** time.

This is because by storing minLateness for the smaller subsets, the time complexity = the number of subsets of all processes which is 2^n. (using property of power set / binomial theorem)
(However if you add in the time it takes to look up the subsets of each size, runtime becomes $O(n2^n)$)

PROBLEM 2:

i) There are no optimization choices to be left here. The longest dependency chain in terms of time is the solution.
Let G be a graph of processes. Have (u, v) be a directed edge from all processes u to all processes v where v is dependent upon u. Let that edge have the weight of u's processing time. Also create a node 'dummy' such that all nodes having no other nodes dependent upon it have an edge directed towards 'dummy' as before. The longest path in this graph can be found using bellman-ford (by negating all edge weights) from all parent (no in-edges) nodes.

ii) Pseudocode:

```
read input into graph G
check feasibility
negate edge weights
run a single Bellman-Ford(G) from all independent nodes
return negative d[] of each process
```

iii) O(VE) because of Bellman-Ford. V is the number of processes, E is the number of dependencies.

iv) There is no optimality choice we are given. Every node can only be scheduled once all of its dependencies are filled so the fastest time that all processes can finish is equal to the time the longest dependency chain (in terms of time) takes to finish. The edge weights of the graph represent time to help find this chain. Therefore the solution is correct.

PROBLEM 3:

i) Whenever a processor frees up, schedule the longest process (among those processes whose dependencies have been met). This is similar to the load balancing heuristic in class.
Alternatively, we can use Problem 2 to find the longest path and put those processes on a processor and then go about scheduling processes from 0. This can be done because the longest path is the BEST case minimization of time so we can just add that to a processor. However, this may not always make the solution more optimal but will reduce the time efficiency, so I have not added this to my solution.

ii)
```
read input into graph G
check feasibility
priority queue processQ for longest process
priority queue processorQ for smallest time
for time t from 0 till all processes are scheduled
   if a process's dependencies are met
         then add to processQ
   while some processor is free at time t
         add processQ.poll() to processorQ.poll()
         re-add the polled processor to Q with updated time
find processor with longest time taken
```

Here each processor has a 'time' variable to indicate how much of it is filled up at that instant.

iii) For p3-testcase.txt:

$T_3 = 29$

$T_{3B} = 34$

$T_N = 29$