Gray Houston
ghousto
0026483532

Note: Throughout this analysis, the variable *n* will always refer to the number of nodes/vertices in the graph and the variable *m* will always refer to the number of edges in the graph.

Project 5 Analysis

1. The asymptotic performance of my implementation of the topological sort algorithm is O(*n* + *m*), where *n* is the number of nodes in the graph and *m* is the number of edges. In my implementation, the source nodes (i.e. nodes with in-degree of 0) are identified when the graph is read in from the file. Therefore, determining the source nodes will not contribute to the algorithm's performance/running time.

   Assuming that the source vertices have previously been identified and stored in a list, my implementation performs a BFS-esque traversal of the graph. When sources nodes are removed from the graph and added to the topological ordering, only that source node's children (i.e. the vertices pointed toward by the source) are checked if they have become "relative" source vertices.

   By only checking a source node's children for new sources and only processing these new sources, each node is accessed and processed exactly once. This traversal of every node exactly once while creating the topological ordering produces an O(*n*) asymptotic performance.

   In addition, the number of nodes traversed by looking at all of a source vertex's children for all relative source nodes is equal to *m*. Proving that every edge is traversed exactly once is slightly non-trivial:

   > All source nodes only contain out-bound edges
   > When a source is removed from the graph, each out-edge to a child is removed
   > Right before a source is deleted, each out-edge is followed to its children
   > Therefore, all the out-edges of a deleted source node are traversed exactly once
   > All of nodes of the graph are eventually removed as "relative" source nodes
   > Therefore, the out-edges of all of the nodes are traversed exactly once
   > The total of all out-edges in the graph is *m*

   Therefore, the total number of node access that occur by checking a source's children for new sources vertices runs asymptotically in O(*m*) time.

   Furthermore, because the O(*n*) traversal to create the topological ordering is not directly related to the O(*m*) traversal used to check a source vertex's children, the performance of these two processes is summed instead of multiplied. Therefore, the overall asymptotic performance of the algorithm is O(*n* + *m*).

2. The asymptotic performance of my implementation of the longest path algorithm runs in O(n + m) time. My algorithm uses a topological ordering and the following analysis will assume that one has already been computed. If the ordering hasn't been created, my algorithm makes one by calling the function described above in question 1. The creating of the topological ordering would incur an additional O(n + m) amount of computation, but does not alter the overall asymptotic performance of the algorithm.

   My implementation works by traversing the nodes in topological order to relax the distances between nodes. This traversal of all n vertices runs in O(n) time asymptotically and, as I will prove below, the relaxation of all edges runs in O(m) time.

   The key to the function of this longest path algorithm is the relaxation of the distances that measure how long it takes to reach an arbitrary node v. Because the edges are not weighted, these distances measure the maximum number of nodes traversed to reach node v from any of the source nodes. These distances are stored in an array where an array index corresponds to the node's id.

   When an arbitrary node v is reached in the topological order, each child of v is accessed to determine if the path to v is longer than any other path that has reached each child vertex so far. Thereby, all of the outbound edges of node v are traversed. In the same manner, all of the out-edges of each node in the digraph is traversed. Because all of the nodes only relax their out-edges once, O(m) node traversals and comparisons occur. Therefore, the relaxation of node distances asymptotically runs in O(m) time.

   Because these two processes are computed independently, the overall performance is the sum the performance of the two tasks. Therefore, the asymptotic running time for the whole algorithm is O(n + m).

3. The asymptotic running time for my implementation of the minimum production span scheduler algorithm is O(n + m). Because my algorithm uses longest path values for vertices, the rest of the analysis of the question assumes that the longest path distances have already been computed. If the longest path values needed to be computed, O(n + m) extra computation would occur, but this would not change the asymptotic performance of the algorithm.

   My implementation is similar to BFS in regards to graph traversal. All the nodes of each step are traversed before moving down to the next step, and nodes can only appear in one step for the entire schedule. Thereby, all n vertices are traversed exactly once when creating the schedule. Therefore, the asymptotic running time to build the schedule is O(n).

   Also, similar to what I have stated in earlier questions, when a vertex is added to the schedule, all of that vertex's outbound edges are followed exactly once. Thereby, all the edges in the graph are traversed exactly once while building the schedule. Therefore, the asymptotic running time to traverse all of the edges is O(*m*).

Therefore, because traversing all *n* elements to build the schedule is not directly computationally related to traversing all of the *m* edges, the performance of these processes should be summed. Therefore, the overall asymptotic performance of the entire algorithm is O(*n* + *m*).

4. The asymptotic performance for my span K stations schedule algorithm implementation is O(*n* + *m*). Similar to the questions above, my implementation of this algorithm uses a topological ordering, and the analysis below assumes the ordering has already been computed. Also, as proven above, having to compute the topological sort would not change the asymptotic performance of the algorithm.

   First, my implementation traverses all *n* nodes in topological order. This traversal asymptotically runs in O(*n*) time.

   While the algorithm runs, a process similar to edge relaxation occurs. For example, take an arbitrary vertex *v* that has just been reached in the topological ordering. To determine which step to which *v* belongs, start looking at the largest step a parent of *v* was placed plus 1. Then, my implementation selects the first step without all the stations used.

   Then all of *v*'s children largest parent step values are relaxed. Similar to the longest path algorithm, the largest parent step value for each child is updated to reflect the largest value seen at that point.

   Through this form of edge relaxation, all *m* outbound edges for all nodes in the digraph are traversed, just the same as the longest paths algorithm. Therefore, the asymptotic performance of relaxing all the edges is O(*m*).

   However, there is one fault in the algorithm. Because we are not allowed to alter the Schedule class, my implementation cannot create a Schedule object with a correct production span until after all of the time values for all *n* nodes have been computed. Therefore, an additional O(*n*) traversal of the vertices is required to physically build the Schedule object and transfer the necessary information. However, because there is already a task that runs proportionally to O(*n*), this extra traversal doesn't asymptotically change the performance.

   Therefore, the performance of the entire algorithm is O(*n* + *m*).

5. An example of a DAG where my *k*-station algorithm fails to produce a schedule with a minimum production span is the example given to us in the 20v25e.txt file. An example of when the algorithm fails is when *k* = 3.

In the example where *k* = 3, there can be no step that uses more than 3 stations. The final output of the algorithm reveals how a minimum production span was failed to be reached:

```
Longest Path = 7
----------------- Results ------------------
The production span is 10
The schedule looks like:
Time
0       2       6       7
1       10      12      13
2       3       16      18
3       1       14      19
4       0
5       4       9
6       5       11
7       8
8       15
9       17
```

As the output above shows, while a minimum production span would run in 8 steps (longest path + 1), the actual production span runs in 10 steps. This sub-optimal result is produced to adhere to the rule that no more than three stations may be used at one time in a step. For the sake of comparison, the output without the *k* = 3 restriction is very different:

```
Longest Path = 7
----------------- Results ------------------
The production span is 8
The schedule looks like:
Time
0       2       6       7       10      12      13      16      18
1       1       3       14      19
2       0
3       4       9
4       5       11
5       8
6       15
7       17
```

Not only is the resulting schedule drastically different, the output is a minimum production span. Therefore, the k-station algorithm fails to produce a min. production span schedule for some values of *k*.

For the first example of the *k*-station algorithm shown above where *k* = 3, the algorithm fails while processing the fourth vertex. At this point in time, three nodes have already been scheduled to run at time 0. Instead of simply scheduling the first three nodes available (like this greedy algorithm would), the vertices scheduled for time 0 should possess the longest paths to the sink.

Thereby, there is still a possibility to create a schedule with a minimum production span. Because the sources with the longest paths to the sink were scheduled at a later step, it became almost impossible to achieve a minimum production span.

6. Similarly to the question above, the graph represented in the 20v25e.txt file can produce a schedule with a minimum production span when *k* is large, such as when *k* = 10. The output for when *k* = 10 is as follows:

```
Longest Path = 7
----------------- Results ------------------
The production span is 8
The schedule looks like:
Time
0       2       6       7       10      12      13      16      18
1       1       3       14      19
2       0
3       4       9
4       5       11
5       8
6       15
7       17
```

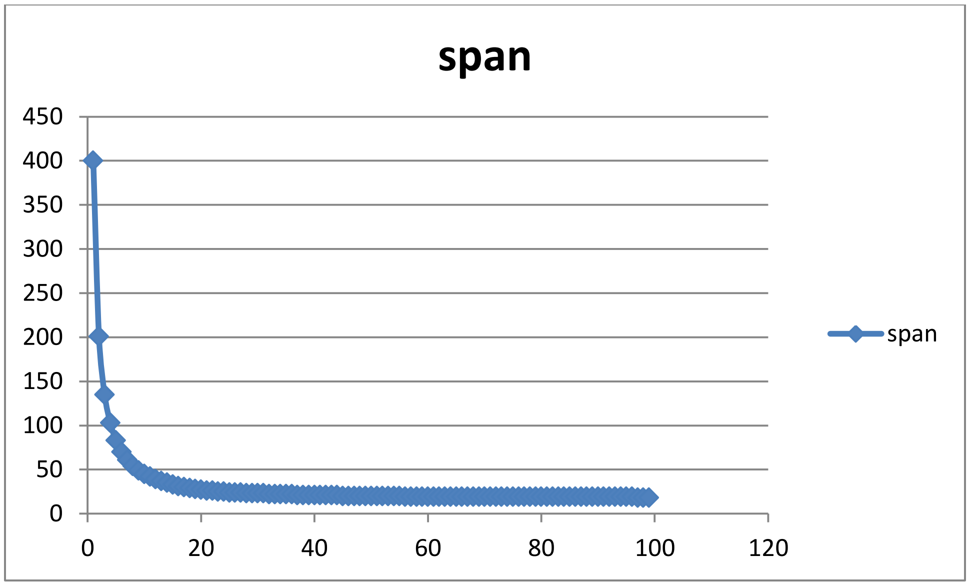
   This output is the same as when there is no restriction on *k*-stations. When the source nodes with the longest paths to the sink are able to be scheduled at time 0, creating a schedule with min. production span is possible.

7. My *k*-stations algorithm for the example described in question 6 would still produce a minimum production span even if the topological ordering was produced using DFS instead of degree-based topological sorting. The algorithm works regardless of the method of topological sort used because the algorithm primarily depends on the edge relaxations to determine which step to schedule a vertex.

   Because the step at which a node is scheduled depends on the step in which its parents are scheduled, the topological ordering has no effect on which step a node is scheduled. While the ordering of the vertices within that step may be different with a DFS topological ordering, the longest path dependencies between the nodes determine which step a node should be scheduled.

   If the topological ordering is not related to which step the vertex is scheduled, then the topological ordering will not effect if a minimum production span can be produced. Therefore, assuming that *k* is large enough to permit the source nodes with the longest path to the sink, using a DFS based topological sort would not cause my *k*-stations scheduling algorithm to fail.

8.



The graph above shows how the size of the production span for the schedule of 400v850e.txt graph decreases in the *k*-stations algorithm as the value of *k* increases from 1 to 99. By the time that *k* reaches the value of 99, the schedule has achieved its minimum production span of 18.

The chart above shows that the production span of the graph follows a drastic power regression. One of the most useful pieces of information that this chart displays is that the production span of the graph strongly adheres to the law of diminishing returns.

The law of diminishing returns states that the benefit of adding an additional unit (e.g. reduction in span by increasing *k* by one) decreases as more units are added. As *k* becomes larger, the reduction in production span produced by increasing *k* by 1 greatly decreases.

For example, the decrease in production span when *k* goes from 1 to 2 is about 200 steps. However, the decrease in production span when *k* jumps from 20 to 40 is only 6 steps.

Furthermore, this chart shows the regions where *k* has strong influence and weak influence. When I say *k* has strong influence, it means that even a small change in *k* will produce a significant effect on the production span. Conversely, when I refer to weak influence, it means that even a large change in *k* will have a small effect on the production span.

The chart above shows that *k* has relatively strong influence in the interval $1 \leq k \leq 20$ and has relatively weak influence for the range $k > 20$. For the range $1 \leq k \leq 20$, the large magnitude of the slope of the graph shows that small change in k would produce strong changes in the production span. Conversely, for the range $k > 20$, the relatively small slope of the graph shows that any change in k produces an insignificant change in the production span.