Assignment 2: Report
IN2010 Høst 2020
siljejka

Complexity

The highest complexity of the methods for an acyclic graph is $O(n^2)$, so this is the complexity of the program when run on an acyclic graph.

Main

The reader runs in $O(n)$ time, where n is the number of lines in the file.

The method then runs graphTraversal.

If there's a cycle it runs printCycle, otherwise it runs topologicalSort, printSchedule, and printProject.

Graph traversal

Let V be the set of tasks and E be the set of dependencies between them.

The method iterates through tasks and runs depthFirst on each unvisited node. Depth first then checks every dependent node and runs depthFirst recursively if it hasn't been visited.

Since it visits every node and edge exactly once the complexity is $O(|V| + |E|)$ or $O(n)$.

Print cycle

For each dependency in the graph it prints them. The maximum number of cycles and dependencies in a complete undirected graph is $\Sigma_{k=3}^{|V|} \frac{1}{2} \binom{|V|}{k}(k-1)!$[1]. For a directed graph this will be double, since a clockwise and counter-clockwise cycle over the same nodes will be distinct.

Since my method also counts 2-node dependencies we'd have to add $\frac{(|V|-1)^2-(|V|-1)}{2}$ to the number.

For each of these it also iterates through each item in each cycle. This will run at most $O(n)$ time.

Not sure what the total complexity for this would be.

Topological sort

The method first makes a copy of the number of predecessors, so that the values of the task nodes themselves won't be changed in the algorithm. This runs in $O(|V|)$ time.

Then we check all nodes for whether they have no conditions, and push all items that have none to the stack. The while loop then runs until the stack is empty, and for each iteration it removes the top item on the stack and checks all it's adjacent nodes for whether this node is its last predecessors. If it is it pushes this node to the stack.

It therefore checks every node and every edge once and has complexity $O(|V| + |E|)$.

[1] According to Wolfram Mathworld on complete graphs.

It then iterates through all the tasks while pushing them in reverse order to the stack. The stack is then used to calculate the latest start by iterating through all the nodes and all it's edges. The complexity is therefore $O(|V| + |E|)$ again.

The total complexity of the method is $O(|V| + |E|)$ or $O(n)$.

Print schedule

The first part iterates over the tasks and adds them to an array, it also has a for loop that adds the number of staff to each time unit in a seperate array. This therefore runs in $O(|V| \cdot n)$ time, or $O(n^2)$.

The next part prints the values stored in the first part by iterating through an array of indexes to check if they contain events. It then iterates over every event at that time and prints it. The number of iterations it'll check is at most the completion time of the task and the number of events it'll have to go through is at most 3n, where n in this case is the number of tasks. The complexity is then O(End time $\cdot 3 \cdot$ no. tasks) or $O(n^2)$

The total complexity is then $O(n^2)$.

Print project

Iterates through every task and prints its information. Runs in $O(n)$.

Graph structure

The graph much be directional and acyclic. If the graph is not directional then nodes will be dependent on each other when adjacent, so that no connected tasks can be completed.

If the graph has any cycles then nodes within the cycle will be dependent on each other, and none of the tasks may be completed.

Graph algorithms

I implemented depth first search to find and store cycles, to do this I used the algorithm shown in lectures and added the boolean variables visited and closed to the task nodes. Visited means the node has been run with the depth first method, while closed means the method has finished for the node, so that all edges from the node has been visited.

Since an adjacent node being visited but not closed means the path has gone back to a parent node, I used this to identify which nodes where part of a cycle.

I also used topological sort from lectures to identify the earliest and latest start times of each task. To find the earliest I first sorted them by execution, since the earliest start time of each node is dependent on when it's predecessors have been completed.

To identify slack I then iterated over a reversely sorted list and found the earliest time each task had to be completed for its successors to complete on time.