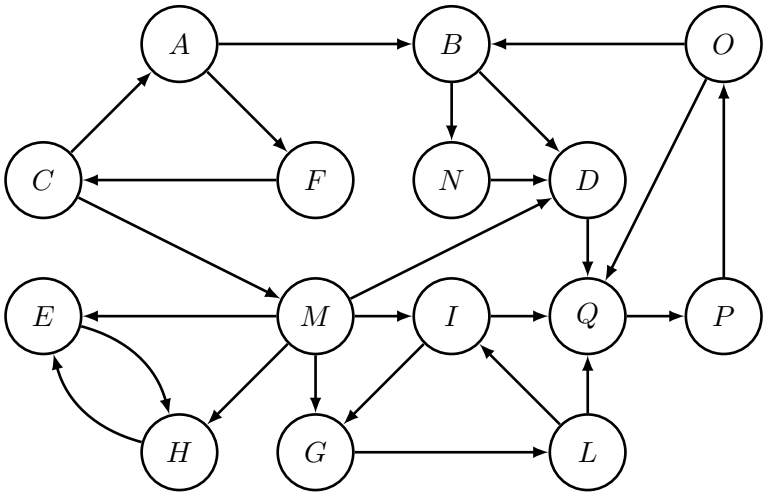
5CCS2FC2: Foundations of Computing II

Tutorial Sheet 5

Solutions

5.1 Consider the following graph $G = (V, E)$ describing a set which tasks that must be completed before or concurrently with which others:



(i) Use the algorithm described in class to identify which tasks must be performed concurrently.

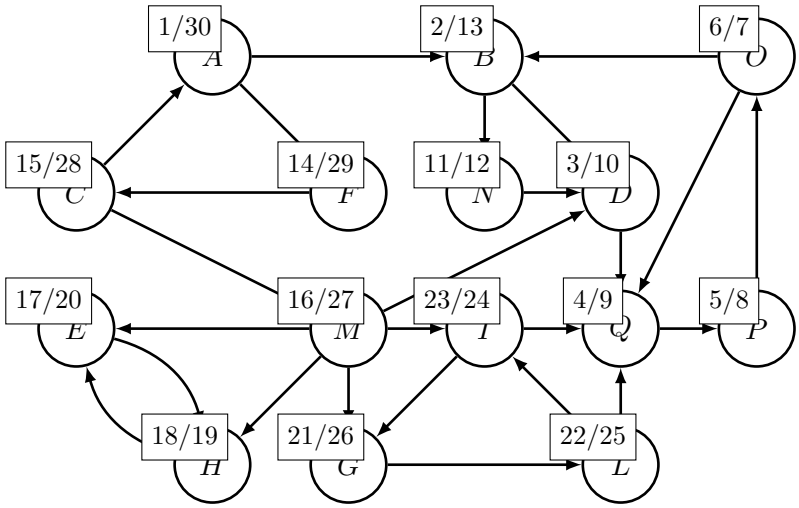
(ii) Draw the component graph G^{SCC}.

(iii) Perform a topological sort on the component grap G^{SCC} to identify an order in which the task can be scheduled?

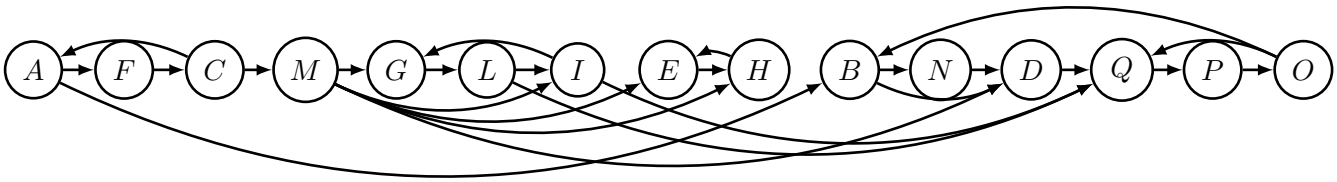
(iv) How many ways can the number of number of strongly connected components change if a single additional edge is added to G?

SOLUTION:

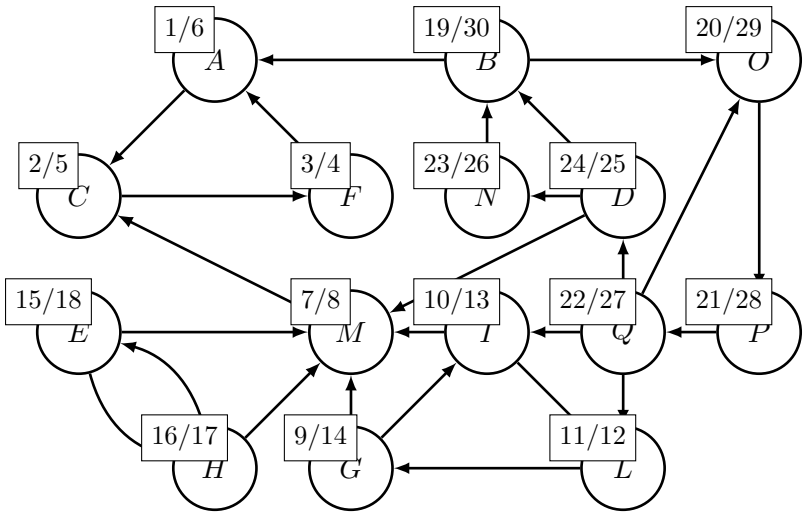
(i) For convenience, we will mark each vertex with a label indicating the *'time'* at which the vertex is first examined/visited and the time at which the vertex is popped from the stack and added to the queue.



This results in the following (almost) topological ordering



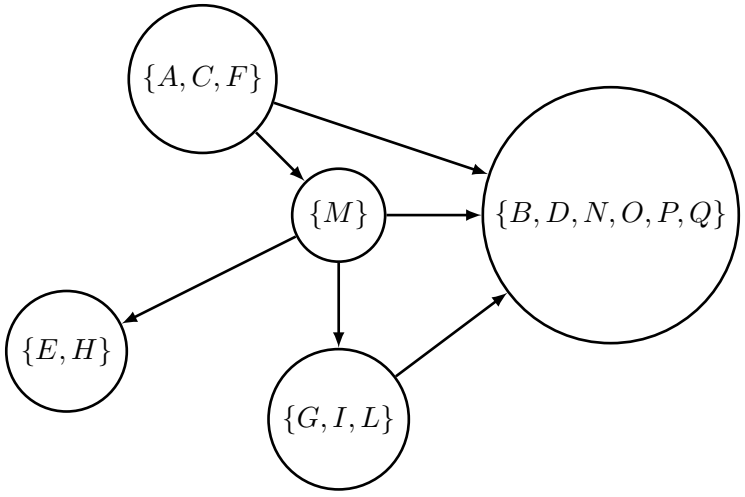
In order to compute the tasks that are required to be concurrent, we must perform a second round of depth-first-search on the transpose graph to identify the strongly connected components, using the above ordering to select the starting verticies.



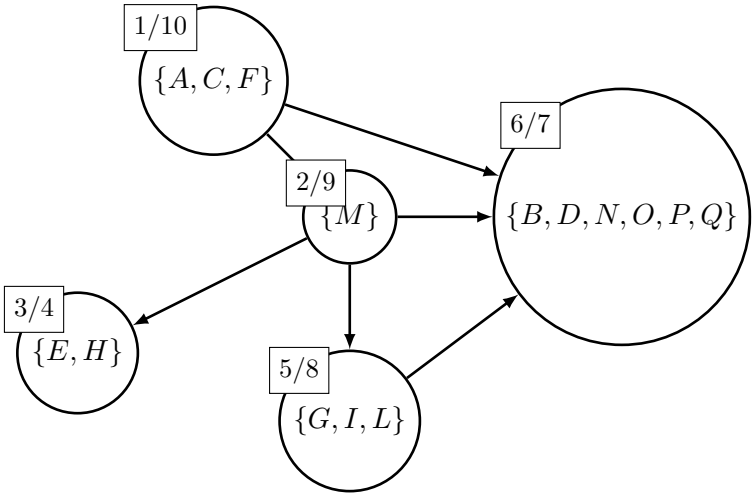
The strongly connected components are obtained whenever we return to the starting vertex and must select a new starting vertex from the queue. Hence, we have the following components:

$$\{A, C, F\}, \qquad \{M\}, \qquad \{E, H\}, \qquad \{G, I, L\}, \qquad \{B, D, N, O, P, Q\}$$

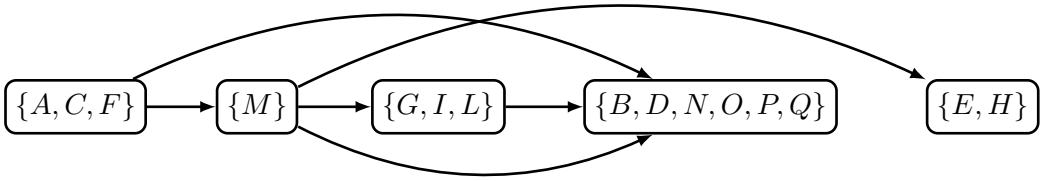
(ii) The component graph of G is given by



(iii) The component graph of G is given by



This results in the following topological ordering



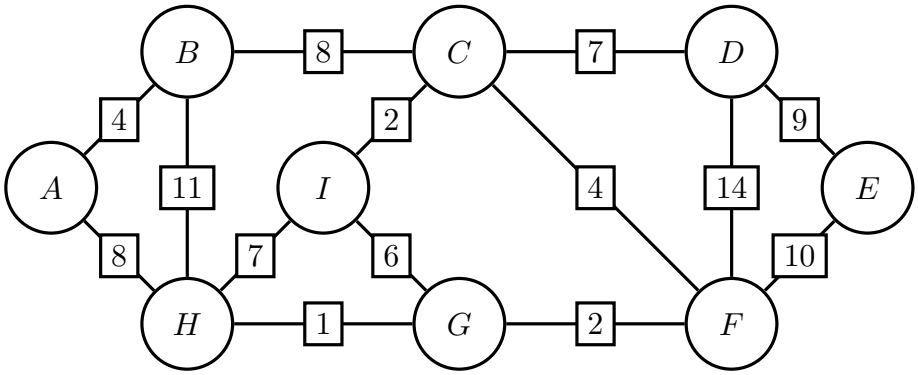
(iv) The number of strongly connected components will only be affected if the edge (u, v) connects vertices in distinct components, where there is already a path between v and u. All components that lie along any path connecting v to u will be merged into a single strongly connected component. In the case of the above graph, we can reduce the number of components in serveral ways; however, there is no single edge that will merge all vertices into a single component.

reduction in number of components	example edge
0	(N, O)
1	(E, M)
2	(I, C)
3	(B, A)
4	impossible

5.2 Consider the following naïve algorithm for the minimum spanning tree problem

> **Step 1)** If the graph contains at most one edge, return this as the minimum spanning tree.
>
> **Step 2)** Otherwise, divide the graph G into two connected subgraphs G_1 and G_2 that differ in size by at most 1 vertex.
>
> **Step 3)** Identify a minimum spanning tree T_1 for G_1 and a minimum spanning tree T_2 for G_2. (Do this by recursively applying this algorithm to each subgraph)
>
> **Step 4)** Connect the two spanning trees T_1 and T_2 with the lightest edge between then.

(i) Apply this algorithm to the following graph:

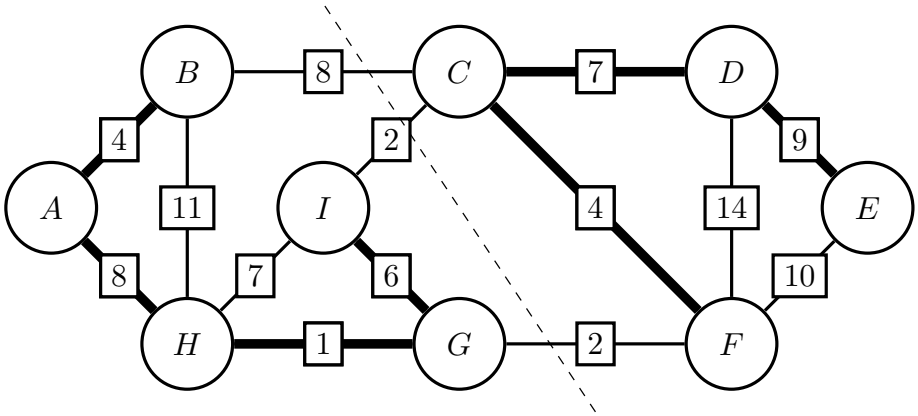


(ii) Does the algorithm always return a spanning tree?

(ii) Does the algorithm always return a *minimum* spanning tree?

SOLUTION:

(i) We can divide the graph into two subgraphs as follows along the dotted line shown below (this cut is arbitrary and may result in different trees depending on which division you make).



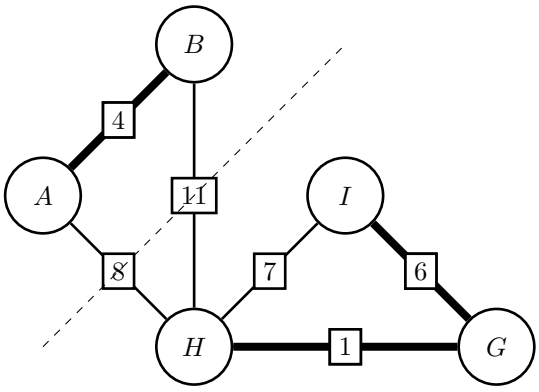
The two spanning trees returned by the algorithm are indicated in bold. There are three edges connectin the two graphs together

$$(B, C), \qquad (C, I), \qquad \text{and} \qquad (G, F)$$

The lightest of which are either (C, I) or (G, F), both with weight 2. Adding either of these will return a spanning tree of weight 41.

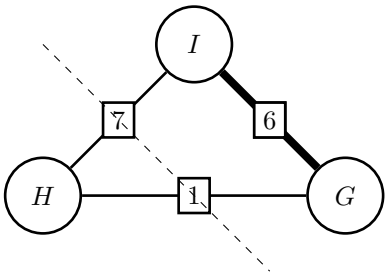
But how do we find the spanning trees for each of the subgraphs in the first place? We must continue subdiving the graph until we find outself with at most one edge which will be their own spanning tree. For the left-most subgraph, we have the following stages:

– We divide the leftmost subgraph into two pieces $\{A, B\}$ and $\{G, H, I\}$, the former of which contains only 1 edge so is returned as its own minimal spanning tree. The lower subgraph must still be subdivided further in the next recursive call.



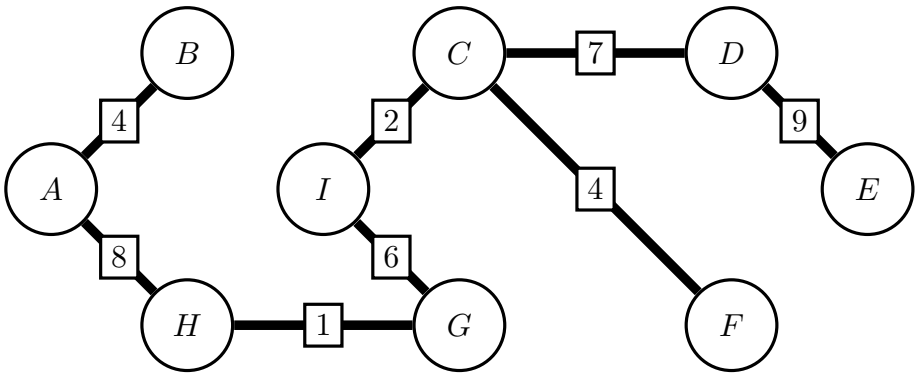
We join these two spanning trees by adding the lightest weight connecting them, which is (A, H) with weight 8.

– Divide the lower subgraph into two pieces $\{H\}$ and $\{G, I\}$, each of which contains at most one edge so they are returned as their own minimal spanning trees.



We join these two spanning trees by adding the lightest weight connecting them, which is (G, H) with weight 1.

The procedure on the rightmost subgraph of G is similar, and the algorithm returns the following proposed spanning tree:

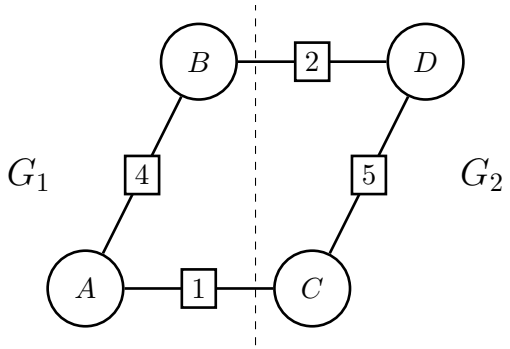


(ii) Yes. We can easily verify that the algorithm returns a spanning tree when given an graph consisting of just a single vertex.

For larger graphs, the algorithm finds two trees T_1 and T_2 which together span all the vertices in the original graph (however, these trees are not connected yet). The algorithm joins the two trees together with a single edge. It is impossible to join two trees with

a single edge and create a cycle since we would need one edge to cross from T_1 to T_2 and a second edge to cross back from T_2 to T_1. Hence, the resulting graph is always a spanning tree.

(iii) No. The algorithm *does not* produce the minimum spanning tree in all cases and depends heavily on how the graph is sub-divided. Consider the following single graph comprising just 4 vertices:



Here, the algorithm would divide the graph into two subgraphs G_1 and G_2 and construct the trees T_1 comprising the edge (A, B) and T_2 comprising the edge (C, D). There would then be joined with the edge (B, D) minimally connecting T_1 and T_2. However, a lighter spanning tree can be obtained by taking (A, C), (A, B), and (B, D).