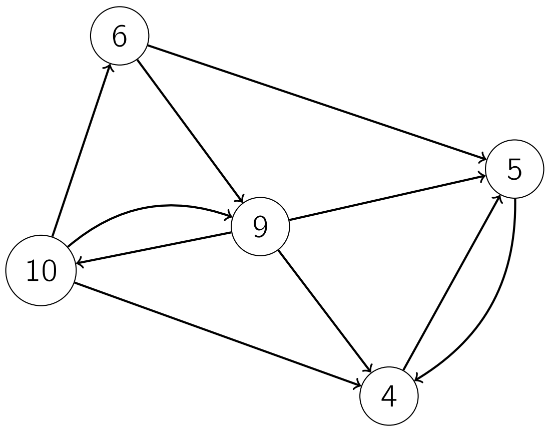
CS 161 (Stanford, Fall 2025) Section 5

1 Algorithm Practice

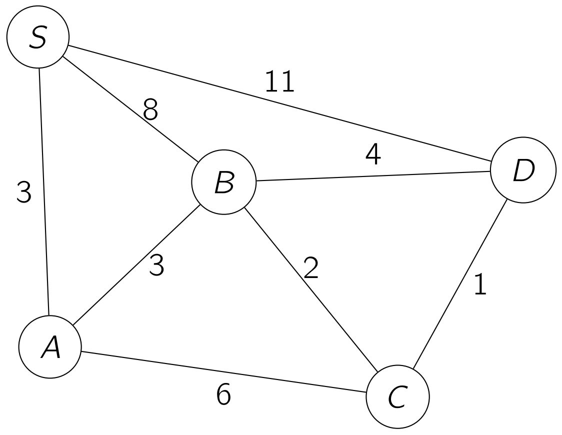
1. In the given graph, the node labels represent the finish times from running depth-first search. Which node would the next DFS call begin from when running Kosaraju's algorithm? Perform this DFS (with edges reversed) to find the find the strongly connected components of the graph.



Solution

The DFS would begin at the node labeled $\boxed{10}$ because it has the highest finishing time. Performing DFS with edges reversed, we explore 10 then 9 and then 6 before backtracking to 10. A second DFS begins at 5 and explores 4. The two SCs found are $\boxed{\{6, 9, 10\}, \{4, 5\}}$.

2. Perform Dijkstra's shortest path algorithm from source S on the graph below, and update the $d[v]$ values for each iteration in the table.



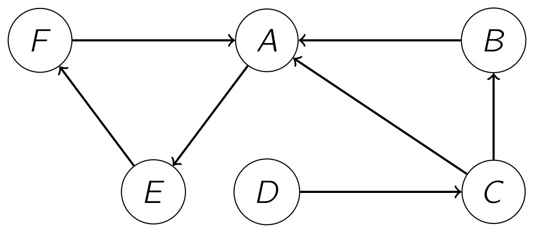
Solution

The completed table is shown below:

Vertex v	$d[v]$	$d[v]$	$d[v]$	$d[v]$	$d[v]$
S	0	0	0	0	0
A	∞	3	3	3	3
B	∞	8	6	6	6
C	∞	∞	9	8	8
D	∞	11	11	10	9

2 Strongly Connected Components

Consider the directed graph below for parts 1 and 2:



(a) How many strongly connected components does this graph have?

Solution

There are $\boxed{4}$ SCCs, $\{A, F, E\}, \{B\}, \{C\}$, and $\{D\}$.

(b) What is the minimum number of directed edges to add to this graph to make all the vertices strongly connected?

Solution

A directed edge from E to D makes all vertices strongly connected so only $\boxed{1}$ edge is needed.

(c) Assume you have two vertices u and v in a directed graph where there exists an edge from u to v. Which one of the following is incorrect about u and v?

(A) u and v can be in the same SCC.

(B) u and v can be in different SCCs.

(C) If u's DFS finish time is less than v's DFS finish time then u and v are in the same SCC.

(D) u's DFS finish time is always greater than v's DFS finish time.

Solution

Answer $\boxed{\text{D}}$ is the only incorrect choice. The edge from A to E in the graph above gives an example of choice A, and the edge from C to A gives an example of choice B. If u and v are in different SCCs then there cannot be a path from v to u. Therefore when u is first visited either v is finished or v is unvisited, in which case it must be visited (and finished) before u can finish. Either way u will have greater finish time than v, so option C is a correct statement. Option D need not be correct. For example, in a graph with just u and v and directed edges both directions, the relative finish time is based on the starting node.

3 Edsger's Apfelstrudel

You are eating at a cozy little restaurant which serves a *prix fixe* menu of $k+1$ courses, with several available choices for each course. Each dish belongs to exactly one course (e.g., risotto can only be ordered as an appetizer, not a main), and you are effectively indifferent between most of the items on the menu (because they are all so tasty), but the main draw of this particular restaurant is that they serve a delicious 'bottomless' dessert: their world-famous Viennese-style apple strudel. They have an unlimited supply of this apple strudel, but each serving will still cost you \$1. The restaurant also has a few interesting rules:

(a) You must finish your current dish before ordering another.

(b) Each dish after the first course depends on what you ordered in the previous course, e.g., you can only order salmon for your main if you ordered a Caesar salad or chicken noodle soup for the previous course. You are told on the menu exactly what these restrictions are before you order anything.

(c) Most importantly, you are not allowed to have their unlimited dessert unless you finish one dish from each of the first k courses!

You are told the cost of each item in each course on the menu, and you plan your meal with a twofold goal: to be able to order the strudel, but also to save as much money as possible throughout the first k courses so that you have more money to spend on the unlimited dessert. Design an algorithm to find the smallest amount of money you can spend on the first k courses and still order the 'bottomless' strudel. If you would like, you may assume the very first course has exactly one choice (e.g., a single complimentary leaf of spinach that costs 0 dollars). Give an $O(k\log(k)+m)$ time algorithm, where m is the number of pairs of dishes for which one dish depends on the other.

Solution

Notice that this problem can be modeled as a shortest-path graph problem with costs on vertices rather than edges. However, we can transform this graph fairly easily into a directed weighted graph.

First, we construct a directed graph in which each node is a dish and edges exist from v_i to v_j if you must finish dish i in order to order dish j.

Next, we augment the edges with weights as follows: for all j, for all (v_i, v_j), the weight of (v_i, v_j) is the cost of v_j.

Finally, we run Dijkstra's algorithm as we saw in class to find the shortest (least cost) path from any dish in the first course to the strudel!

An equivalent formulation of this algorithm is to skip the edge-weighting step and instead use a modified version of Dijkstra's algorithm in which path lengths are given by the sum of the costs of the vertices along that path.

Aside: note that without the dependencies caveat, a naive greedy algorithm solves this problem.

4 Finding a target vertex

Let $G = (V, E)$ be a directed acyclic graph. Say that a vertex v is a *target* if, for all $u \in V$, there is a path from u to v. Given an algorithm that runs in time $O(n + m)$ that either finds a target vertex, or else returns `None` if no target vertex exists.

Solution

Run the topological sorting algorithm from class to find an ordering $v_1, v_2, \ldots, v_n$ on the vertices. Then reverse all the edges of the graph and do a breadth-first search starting from v_n, keeping track of how many distinct vertices you visit. If you visit all n vertices, return v_n. Otherwise, return `None`. This takes $O(n + m)$ time.
The reason this works is that if there is a target vertex, then v_n must be it. Indeed, if v_t were a target vertex for $t < n$, then there would be a path from v_n to v_t, which would violate the toposort order. So v_n is the only candidate. Then we test to see if v_n is indeed a target by doing BFS (on the reversed graph) to see if every vertex could have reached v_t.
Alternative solution. Note that alternatively, there is a target vertex if and only if there is exactly one node with no outgoing edges.

5 High Speed Cable Internet

Algorithmia, an internet service provider, has a new high speed cable internet technology that will require new cable installation. They will install these new cables on the currently existing network of cables but it will be costly.

This can be modelled with a weighted undirected graph $G = (V, E)$ with non-negative edge weights. The nodes represent neighborhoods, edges represent the existing cables between the neighborhoods, and edge weights represent the cost to install a new cable.

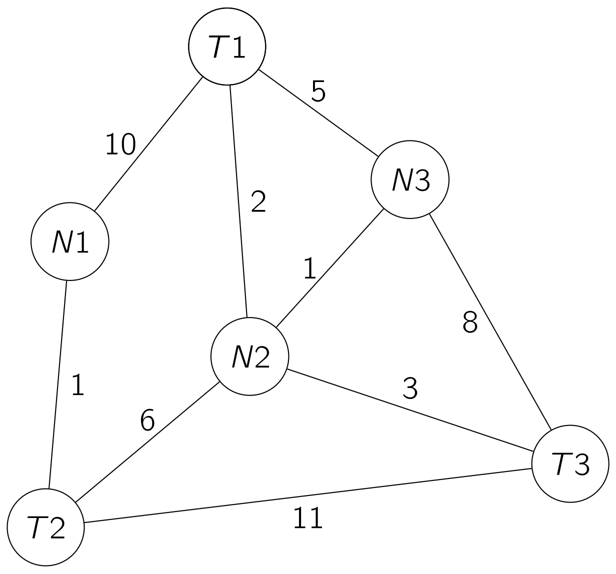
Because of limited resources and to minimize costs, Algorithmia has chosen to start with the neighborhoods with highest demand for this high speed internet access that will lead to the highest profit, creating a set $T \subset V$ of terminals which includes the neighborhood with Algorithmia's headquarters along with the high demand neighborhoods.

To connect all these neighborhoods, we can model this with a *Steiner tree*, a tree (aka graph with no cycles) that contains all of the terminals and possibly some other vertices. Algorithmia wants to find the minimum weight Steiner tree to find the lowest cost to install new cables that connect Algorithmia's headquarters and the high demand neighborhoods.

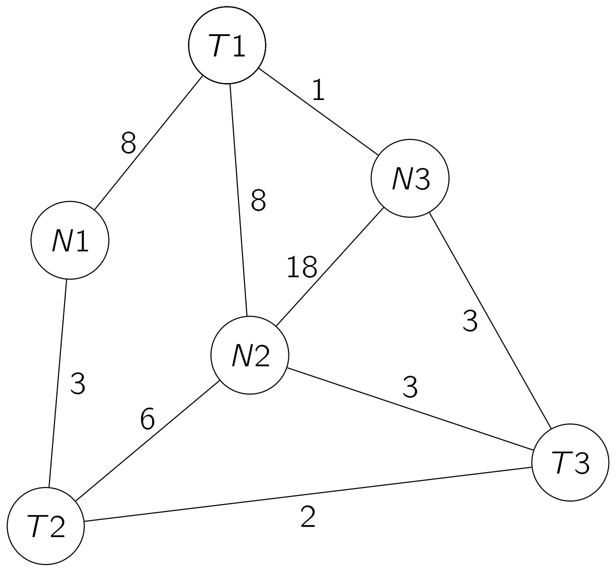
(a) If there are only two terminals (Algorithmia's headquarters and another high demand neighborhood), give an $O(n \log(n) + m)$-time algorithm for finding a minimum weight Steiner tree. [**We are expecting:** *English description and brief running time analysis*]

> Solution
>
> Run Dijkstra from one terminal to the other. This will return the lowest weight path that connects the two terminals, which will be the minimum Steiner Tree.

(b) For terminals $T1, T2, T3$, **draw** the minimum weight Steiner tree in each of the following graphs: [**We are expecting:** *A drawing of the Steiner trees.*] (You can use copy+paste the tikz code on tex and delete the lines corresponding to edges that do not participate in the Steiner tree.)

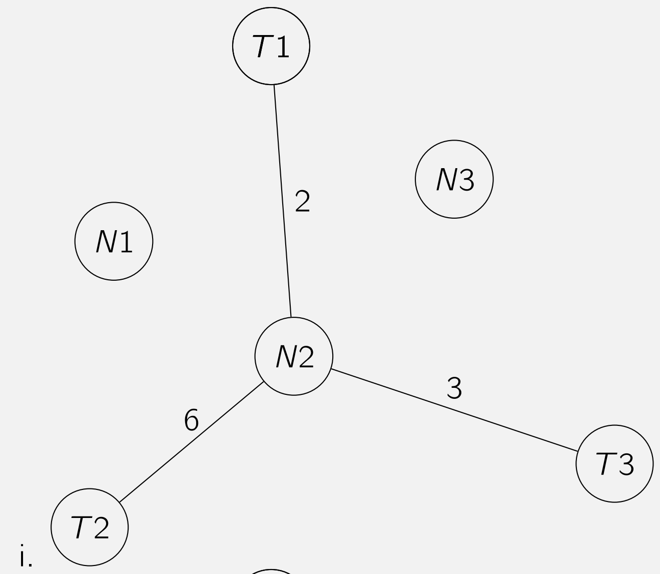


i.

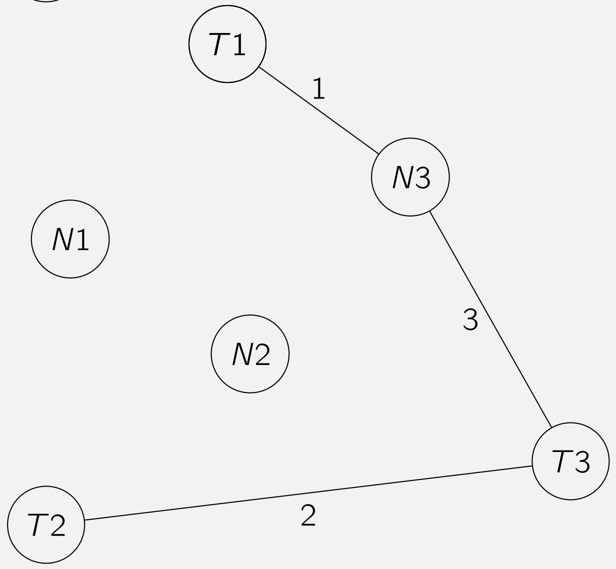


ii.

Solution



i.



ii.

(c) **Bonus:** Give an $O(n^2 \log(n) + nm)$-time algorithm for finding a minimum weight Steiner tree with **three terminals**. [**We are expecting:** *English description, pseudocode, and running time analysis*]

Solution

We will guess the "middle" vertex of the Steiner Tree and calculate Dijkstra from this point to all the other terminals. We will repeat this for all possible nodes on the graph and return tree resulting from the smallest choice of a "middle."

```
Three_Terminal_Steiner(Graph G, Terminals T):
   minimum_tree_cost = inf;
   final_tree_edges = {};

   for all nodes n:

      all_shortest_paths = Dijkstra(G, n);
      local_cost = 0;
      local_edge_set = {};

      for terminal in T:
         // Get cost for shortest path from middle node
         // to each terminal;
         local_cost += all_shortest_paths[terminal].cost;
         local_edge_set += all_shortest_paths[terminal].path;

      if local_cost < minimum_cost:
         minimum_cost = local_cost;
         final_tree_edges = local_edge_set;

   return final_tree_edges, minimum_cost;
```

Runtime: This method will run for all n nodes, and will run Dijkstra's for each node. Since Dijkstra's Algorithm runs in $O(n \log(n) + m)$ and we are running it for all n nodes, the total runtime is $O(n^2 \log(n) + nm)$.