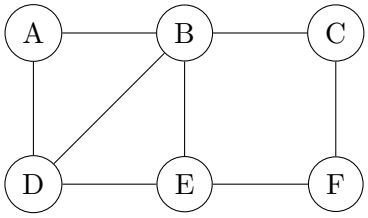
CS 170 DIS 03

Released on 2018-02-07

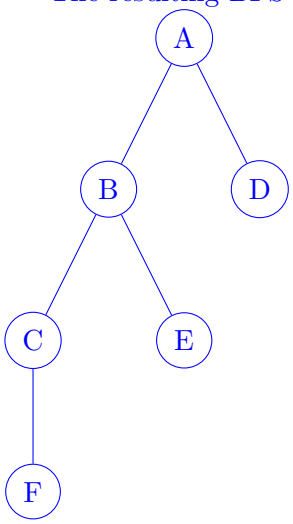
1 Breadth-First Search



Run breadth-first search on the above graph, breaking ties in alphabetical order (so the search starts from node A). At each step, state which node is processed and the resulting state of the stack. Draw the resulting BFS tree.

Solution:

The resulting BFS tree:



The node processed at each step and the resulting stack:

Node Processed	Stack
	{A}
A	{B, D}
B	{D, C, E}
D	{C, E}
C	{E, F}
E	{F}
F	{}

2 Dijkstra's Algorithm Fails on Negative Edges

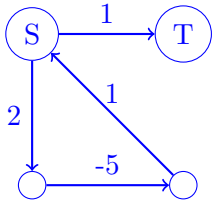
Draw a graph with five vertices or fewer, and indicate the source where Dijkstra's algorithm will be started from.

1. Draw a graph with at least one negative weight edge for which Dijkstra's algorithm produces the wrong answer.

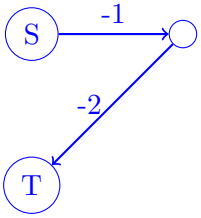
2. Draw a graph with at least two negative weight edge for which Dijkstra's algorithm produces the correct answer.

Solution:

1. Dijkstra's algorithm fails when negative weight cycles are present. For example, Dijkstra's algorithm returns a path from S to T of weight 1 in the following graph:



2. Dijkstra's algorithm works on DAGs because no negative cycles are present. For example, Dijkstra's with product the correct answer of -3 on the following graph:



3 Fixing Dijsktra's Algorithm with Negative Weights

Dijkstra's algorithm doesn't work on graphs with negative edge weights. Here is one attempt to fix it:

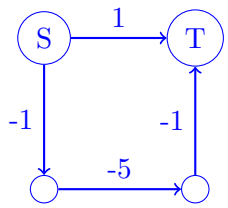
1. Add a large number M to every edge so that there are no negative weights left.

2. Run Dijkstra to find the shortest path in the new graph.

3. Return the path Dijkstra found, but with the old edge weights (i.e. subtract M from the weight of each edge).

Show that this algorithm doesn't work by finding a graph for which it must give the wrong answer.

Solution: The above algorithm doesn't work when the actual shortest path has more edges than other potential shortest paths. In this case, the paths with more edges have their weights increased more than the path with fewer edges. We can see this in the following counterexample:



The shortest path is "down-right-up" (weight -7). After adding $M = 5$ to each edge, we increase the actual shortest path by fifteen 15. The path "right" only increases by 5 and so the algorithm returns this path as the shortest path.

4 Bounded Bellman-Ford

Modify the Bellman-Ford algorithm to find the weight of the lowest-weight path from s to t with the restriction that the path must have at most k edges.

Solution: The obvious instinct is to run the outer loop of Bellman-Ford for k steps instead of $|V| - 1$ steps. However, what this does is to guarantee that all shortest paths using at most k edges would be found, but some shortest paths using more that k edges might also be found. For example, consider a path on 10 nodes starting at s and ending at t, and set $k = 2$. If Bellman-Ford processes the vertices in the order of their increasing distance from s (we cannot guarantee beforehand that this will **not** happen) then just one iteration of the outer loop finds the shortest path from s to t, which contains 10 edges, as opposed to our limit of 2. We therefore need to limit Bellman-Ford so that results computed during a given iteration of the outer loop are not used to improve the distance estimates of other vertices during the **same** iteration.

We therefore modify the Bellman-Ford algorithm to keep track of the distances calculated

in the previous iteration.

Input: Directed Graph $G = (V, E)$; edge lengths l_e on the edges, vertex $s \in V$, and an integer $k > 0$.

Output: For all vertices $u \in V$, $dist[u]$, which is the length of path of lowest weight from s to u containing at most k edges.

foreach $v \in V$ **do**
    `dist`$[u] = \infty$;
    `new-dist`$[u] = \infty$;
end

`dist`$[s] = 0$;
`new-dist`$[s] = 0$;

for $i = 1$ **to** k **do**
    foreach $v \in V$ **do** `// Copy new-dist into previous-dist`
        `previous-dist`$[v] =$ `new-dist`$[v]$;
    end
    foreach $e = (u, v) \in E$ **do**
        `new-dist`$[v] = \min($`new-dist`$[v],$ `previous-dist`$[u] + l_e)$;
    end
end

Algorithm 1: Modified Bellman-Ford

Assume that at the beginning of the ith iteration of the outer loop, `new-dist`$[v]$ contains the lowest possible weight of a path from s to v using at most $i - 1$ edges, for all vertices v. Notice that this is true for $i = 1$, due to our initialization step. We will now show that the statement also remains true at the beginning of the $(i + 1)$th iteration of the loop. This will prove the correctness of the algorithm by induction. We first consider the case where there is no path from s to v of length at most i. In this case, for all vertices u such that $(u, v) \in E$, we must have `new-dist`$[u] = \infty$ at the beginning of the loop. Thus, `new-dist`$[v] = \infty$ at the end of the loop as well. Now, suppose that there exists a path (not necessarily simple) of length at most i from s to v, and consider such a path of smallest possible weight w. We want to show that `new-dist`$[v] = w$.

Let u be the vertex just before v on this path. By the induction hypothesis, at the end of the loop on line 1, `previous-dist`$[u]$ stores the weight of the lowest weight path of length at most $i - 1$ from s to u, so that when the edge (u, v) is proceed in the loop on line 1, we get `new-dist`$[v] \leq w$.

Now, we observe that at the end of the loop on line 1, we have

$$\texttt{new-dist}[v] = \min\left(\texttt{previous-dist}[v], \min_{u:(u,v)\in E}\left(\texttt{previous-dist}[u] + l_{(u,v)}\right)\right).$$

Note that by the induction hypothesis, each term in the minimum expression represents the length of a (not necessarily simple) path from s to v of length at most i. Thus, in particular, none of these terms can be smaller than w, so that `new-dist`$[v] \geq w$. Combining with `new-dist`$[v] \leq w$ obtained above, we get `new-dist`$[v] = w$ as required.