1. Shortest paths, a saga in 5 parts.

Suppose you’re using Dijkstra’s algorithm starting from some source vertex s. The table on the right shows the shortest paths tree (edgeTo[] and distTo[]) immediately after vertex 4 has been relaxed.

edge weight

3.0

|  |  |  |
| --- | --- | --- |
| v | distTo[] | edgeTo[] |
| 0 |  | *null* |
| 1 | 7.0 | 5 |
| 2 | 13.0 | 3 |
| 3 | 0.0 | *null* |
| 4 | 10.0 | 7 |
| 5 | 3.0 | 3 |
| 6 | 12.0 | 1 |
| 7 | 8.0 | 3 |

1.0

1.0

5.0

2.0

17.0

13.0

3.0

8.0

1.0

4.0

-11.0

2.0

1. Give the order in which the first 5 vertices were deleted from the priority queue and relaxed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **3** | **5** | **1** | **7** | 4 |

1. Fill in the table below to reflect the new shortest paths tree (by changing edgeTo[] and distTo[]) after the next vertex is relaxed. Only fill in those entries which change from the values shown in the table above.

|  |  |  |
| --- | --- | --- |
| v | distTo[] | edgeTo[] |
| 0 |  | *null* |
| 1 | 7.0 | 5 |
| 2 | 13.0 | 3 |
| 3 | 0.0 | *null* |
| 4 | 10.0 | 7 |
| 5 | 3.0 | 3 |
| 6 | 12.0 | 1 |
| 7 | 8.0 | 3 |

|  |  |  |
| --- | --- | --- |
| v | distTo[] | edgeTo[] |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 | **1.0** | **6** |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |

1. As you are no doubt aware, Dijkstra’s algorithm is not guaranteed to provide the correct shortest paths tree when a graph contains a negative weight. For the graph above, it will not.

Suppose we create a new graph that is different only in the weight between . Provide a new weight for the edge that ensures that Dijkstra’s provides the correct shortest paths tree (assuming all other weights in the graph stay the same). Include a brief explanation of why this new edge weight works.

**Choosing any weight > 10 ensures that vertex 6 relaxes before vertex 4, and thus edge is guaranteed to get utilized. Alternate answer was to choose any weight 2, in which case the edge is irrelevant.**

1. A modified version of Dijkstra’s algorithm with two additional lines of code is shown below (annotated in bold). Given a graph G for which Dijkstra’s algorithm returns a correct result, will this version of Dijkstra’s algorithm always return the correct result G? Give an intuitive reason for your answer (you do not need to provide a full proof).

public DijkstraSP(EdgeWeightedDigraph G, int s) {

distTo = new double[G.V()];

edgeTo = new DirectedEdge[G.V()];

for (int v = 0; v < G.V(); v++)

distTo[v] = Double.POSITIVE\_INFINITY;

distTo[s] = 0.0;

*// relax vertices in order of distance from s*

pq = new IndexMinPQ<Double>(G.V());

pq.insert(s, distTo[s]);

while (!pq.isEmpty()) {

int v = pq.delMin();

for (DirectedEdge e : G.adj(v))

relax(e);

**for (DirectedEdge e : G.edges()) // G.edges() returns an Iterable of**

**relax(e); // every DirectedEdge in G.**

}

}

**Works fine: Relaxing additional edges can never ruin a correct set of shortest paths.**

1. Which of the following algorithms are guaranteed to give the correct shortest paths on the graphs listed?

--**Y**-- For a graph with no negative cycles: Relax all vertices E times.

--**Y**--- For a graph with no negative cycles: Relax all vertices. Repeat E times.

--**N**--- For a directed acyclic graph: Relax all vertices once.

--**N**--- For a graph with no negative cycles: Relax all vertices with in-degree 0, then relax all vertices with in-degree 1, and so on.

**The first algorithm is like Bellman-Ford but does way too much work, so it’ll definitely work. The second algorithm is Bellman-Ford. The third algorithm is like the topological sort algorithm but it might do things in a bad order (for example, relaxing the source vertex last). The last algorithm might fail on a graph like A->B->C.**

1. Suppose you know the MST of a weighted graph G. Now a new edge v-w of weight c is inserted into G to form a new graph G’. Design a algorithm to determine if the MST in G is also an MST in G’. You may assume all edge weights are distinct.

**For simplicity, we’ll assume all edge weights are distinct.**

**Find the unique path between v and w in the MST. This takes O(V ) time using BFS or DFS because there are only V − 1 edges in the MST subgraph. We claim that the MST in G is the same as the MST in G0 if and only if every edge on the path has length less than c.**

**• If any edge on the path has weight greater than c, we can decrease the weight of the MST by swapping the largest weight edge on the path with v-w. Hence weight of the MST for G0 is strictly less than the weight of the MST for G.**

**• If the weight of v-w is larger than any edge on the path between v and w, then the cycle property asserts that v-w is not in the MST for G0 (because it is the largest weight edge on the cycle consisting of the path from v to w plus the edge v-w). Thus, the MST for G is also the MST for G0.**

1. Given an edge-weighted graph G and an edge e, design a linear-time algorithm to determine whether e appears in an MST of G. Assume G is connected and all edge weights are distinct.

**Remove all edges of weight greater than e. Find the connected components in this new graph. The cut property tells us that e is in the MST if and only if v and w are in different components.**

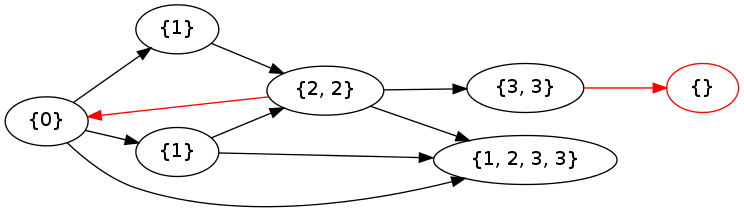
1. Let G=(V,E) be an unweighted, directed graph. Let s and t be two vertices of G. Design an algorithm with O(V+E) worst-case running time that finds the number of distinct shortest paths (distinct paths may share some but not all vertices) from s to t. You may assume there are no parallel edges or self-edges.

Since we’re looking for shortest paths, it’s pretty clear we’re going to need to use some sort of BFS.

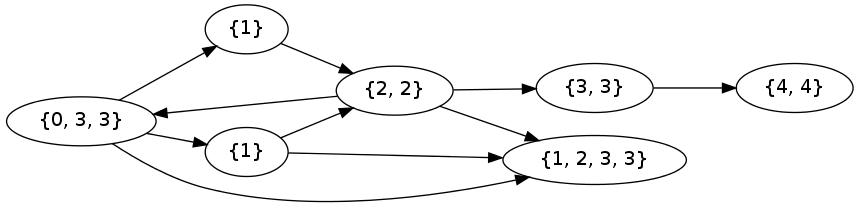
The first insight in this problem is that we don’t need to worry at all about cycles. Since we’re dealing with shortest paths, we know that any path involving a cycle will not count as a shortest path.

Another useful insight is that if we’re using BFS, every incoming edge to vertex X that could possibly be part of a shortest path to X will be processed before vertex X is dequeued. This means that we can move on from vertex X as soon as X is dequeued (i.e. there’s no need to wait for every incoming edge, so we can proceed in normal BFS order).

Given the two insights above, we have a pretty natural (but possibly slow) algorithm that we can use as a starting point. We simply give every vertex an empty Bag, where each entry in the Bag represents the length of a particular path to that vertex. We start the source vertex off with a 0 in its bag, and every other bag starts empty. When an edge is processed (using BFS), the list inside is incremented by 1 and appended to ’s list. For example, consider the graph below, where vertices in red have not yet been dequeued, and edges in red have not yet been processed.

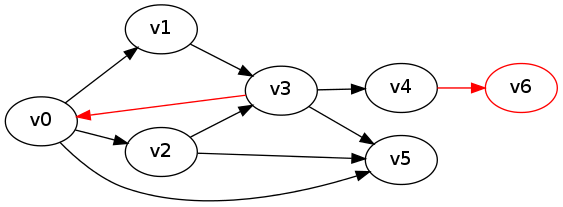


When this algorithm completes, we’re left with the graph shown below. To determine the number of shortest paths to a node, we simply count the number of occurrences of the minimum inside the node.



This algorithm is too slow, because in a fully connected graph, we’d have to build V lists of length V, which is nonlinear. To improve performance, we can take advantage of the fact that we don’t really need to keep anything other than the number of shortest paths for each node. We can accomplish this by adding a third array to BFS called pathCount of length V.

In this improved algorithm we start by setting pathCount inside each node to 0. When processing an edge , if distTo[b] = distTo[a] + 1, we increment pathCount[b] by pathCount[a] (since vertex a provides a new set of shortest distTo[a]). If distTo[b] > distTo[a] + 1, we’ve found a new shortest path, and set pathCount[b] to pathCount[a]. If distTo[b] < distTo[a] + 1, we do nothing since the path(s) under consideration is too long to be considered. Think of distTo[b] as the “old path”, and distTo[a] + 1 as the “new path”.



For example, for the first graph above, our graph state would be given by:

|  |  |  |  |
| --- | --- | --- | --- |
|  | distTo | edgeTo | pathCount |
| v0 | 0 | (don’t care) | 1 |
| v1 | 1 | (don’t care) | 1 |
| v2 | 1 | (don’t care) | 1 |
| v3 | 2 | (don’t care) | 2 |
| v4 | 3 | (don’t care) | 2 |
| v5 | 1 | (don’t care) | 1 |
| v6 |  | (don’t care) | 0 |

As an example, when the edge from is processed, the algorithm will see that distTo[v6] > distTo[v4] + 1, thus pathCount[v6] will be set equal to pathCount[v4].

1. Suppose we want to find the second shortest paths from s to every other vertex. Can the solution be represented by a tree?

**No. Imagine a graph with edges A->B, A->C, B->C where all weights are 1. There is no second shortest path to B, but the second shortest path to C is A->B->C. If we try to do something like the shortest paths tree (i.e. a distTo[] and edgeTo[] array), then we’ll run into trouble since this data structure would imply the path to B is a second shortest path.**