**Graphs**

*Shortest Paths, Dijkstra, Bellman-Ford*

1. Suppose that we convert an EdgeWeightedGraph into an EdgeWeightedDigraph by creating two DirectedEdge objects in the EdgeWeightedDigraph (one in each direction) for each Edge in the EdgeWeighted Graph and then use the Bellman-Ford algorithm. Exaplin why this approach fails spectacularly.

**Solution**: This can introduce negative cost cycles even if the edge-weighted graph does not contain them. Nese do te duhej t’i relaksonim te gjitha lidhjet duke perfshire dhe lidhjet paralele me drejtime te kunderta, queue nuk do boshatisej kurre per shkak te ndryshimit te vazhdueshem te distancave dhe algoritmi nuk do mbyllej me V-1 iterime. Kjo per shkak se krijohet nje cikel negative edhe pa patur pesha negative psh.

1. What happens if you allow a vertex to be enqueued more than once in the same pass in the Bellman-Ford algorithm?

**Answer**: Do rritej kompleksiteti ne kohe i algoritmit ne varesi te sa here ajo futet ne queue. (shtohet me EV kompleksiteti per cdo shtim te tepert ne queue)The running time of the algorithm can go exponential. For example, consider what happens for the complete edge-weighted digraph whose edge weights are all -1. (pra nqs kemi lidhje te perseritura)

1. Does Dijkstra’s algorithm work with negative weights?

**Answer**: Yes and no. There are two shortest paths algorithms known as Dijkstra’s algorithm, depending on whether a vertex can be enqueued on the priority queue more than once. When the weights are nonnegative, the two versions concide (as no vertex will be enqueued more than once). The version implemented in DijkstraSP.java (which allows a vertex to be enqueued more than once) is correct in the presence of negative edge weights (but no negative cycles) but its running time is exponential in the worst case. If we modify DijkstraSP.java so that a vertex cannot be enqueued more than once (e.g using a marked[ ] array to mark those vertices that have been relaxed), then the algorithm is guaranteed to run in ElogV time but it may yield incorrect results when there are edges with negative weights. (pra, nese futet me teper se 1 here ne queue nje nyje, mund te marrim rezultat te sakte – kjo jep kompleksitet V2 – ne te kundert nuk jep rezultat te sakte)

1. True or false. Adding a constant to every edge weight does not change the solution to the single-source shortest-paths problem

**Solution**. False. Psh: nese kemi nje graf me lidhje negative dhe gjejme lidhjen me te vogel (pra ate qe eshte me negativja nga te gjitha), dhe pastaj ia shtojme si vlere absolute te gjitha peshave te tjera, atehere e transformojme grafit ne nje graf me pesha jo negative. Kjo menyre nuk funksionon sepse shortest paths e reja nuk kane lidhje me shortest paths e vjetra. Sa me shume lidhje te permbaje nje rruge, aq me teper eshte e penalizuar nga ky ndryshim. The problem is that different paths from one vertex to another might not have the same number of edges. If we add some number to the length of each edge, then the lengths of different paths can increase by different amounts, and a shortest path in the new graph might be different than in the original graph.

1. All-pairs shortest paths on a line. Given a weighted line-graph (undirected connected graph, all vertices of degree 2, except two endpoints which have degree 1), devise an algorithm that preprocesses the graph in linear time and can return the distance of the shortest path between any two vertices in constant time. (**Zgjidhur – AllPairsSPLine.java**)

Partial solution. Find a vertex s of degree 1 and run bfs (ose dfs) to find the order in which the remaining vertices appear. Then, compute the length of the shortest path from s to v for each vertex v, say dist[v]. The shortest path between v and w is |dist[v] – dist[w]|.

1. Monotonic shortest path. Given an edge-weighted digraph, find a monotonic shortest path from s to every other vertex. A path is monotonic if the weight of every edge on the path is either strictly increasing or strictly decreasing. (**Zgjidhur – MonotonicShortestPath.java**)

Partial solution. Relax edges in ascending order and find a best path; then relax edges in descending order and find a best path.

1. Lazy Implementation of Dijkstra’s algorithm. Develop an implementation LazyDijkstraSP.java of the lazy version of Dijkstra’s algorithm that is described in the text. (**Zgjidhur – LazyDijkstra.java**)
2. Bellman-Ford queue never empties. Show that if there is a negative cycle reachable from the source in the queue-based implementation of the Bellman-Ford algorithm, then the queue never empties.

Duke konsideruar nje graft shembull me lidhje v->w w->z z->v. Ku peshat e lidhjeve jane: v->w:1; w->z:1; z->v: -3

Solution: Consider a negative cycle and suppose that distTo[w]<=distTo[v]+length(v,w) for all edges on cycle W. Summing up this inequality for all edges on the cycle implies that the length of the cycle is nonnegative. (nuk do boshatisej kurre sepse do vazhdonte cikli i pafundem)

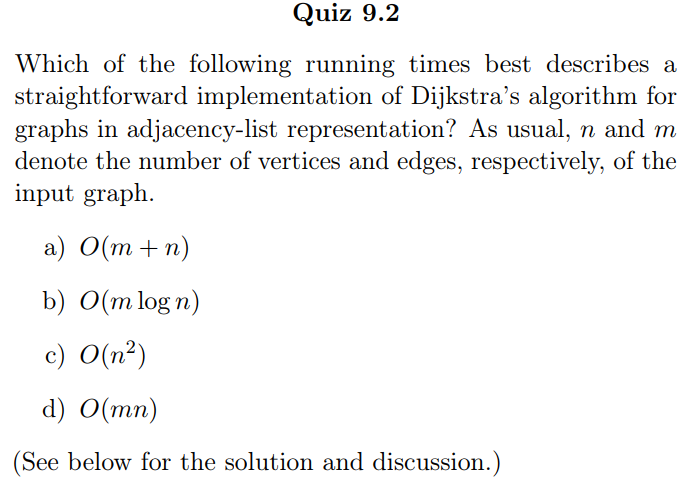
1. Bellman-Ford negative cycle detection. Show that if any edge is relaxed during the Vth pass of the generic Bellman-Ford algorithm, then the edgeTo[] array has a directed cycle and any such cycle is a negative cycle.
2. Replacement paths. Given an edge-weighted digraph with nonnegative weights and source s and sink t, design an algorithm to find the shortest path from s to t that does not use edge e for every edge e. The order of growth of your algorithm should be EVlogV.
3. Shortest path with the ability to skip one edge. Given an edge-weighted digraph with nonnegative weights, design an ElogV algorithm for finding the shortest path from s to t where you have the option to change the weight of any one edge to 0.

Solution: Compute the shortest path from s to every other vertex; compute the shortest path from every vertex to t. For each edge e = (v, w), compute the sum of the length of the shortest path from s to v and the length of the shortest path from w to t. The smallest such sum provides the shortest such path.

1. Shortest paths in undirected graphs. Write a program [DijkstraUndirectedSP.java](https://algs4.cs.princeton.edu/44sp/DijkstraUndirectedSP.java.html) that solves the single-source shortest paths problems in undirected graphs with nonnegative weights using Dijkstra's algorithm.
2. The diameter of a digraph is the length of the maximum-length shortest path connecting two vertices. Write a DijkstraSP client that finds the diameter of a given EdgeWeightedDigraph that has nonnegative weights.
3. What happens to Bellman-Ford if there is a negative cycle on the path from s to v and then you call pathTo(v)?

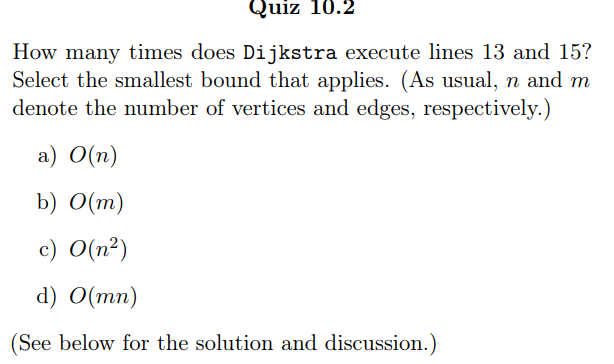
Answer: Do kishim cikel te pafundem.

1. Negative cycle detection. Suppose that we add a constructor to the Bellman-Ford algorithm that differs from the constructor given only in that it omits the second argument and that it initializes all distTo[] entries to 0. Show that, if a client uses that constructor, a client call to hasNegativeCycle( ) returns true if and only if the graph has a negative cycle (and negativeCycle( ) returns that cycle).

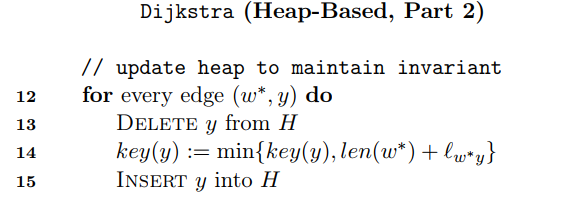


Correct answer: (d).

A straightforward implementation keeps track of which vertices are in X by associating a boolean variable with each vertex. Each iteration, it performs an exhaustive search through all the edges, computes the Dijkstra score for each edge with tail in X and head outside X (in constant time per edge), and returns the crossing edge with the smallest score. After at most n-1 iterations, the Dijkstra algorithm runs out of new vertices to add to its set X. Because the number of iterations is O(n) and each takes time O(m), the overall running time is O(mn).



Correct answer: (b).



In one iteration of the main loop, these two lines might be performed as many as n-1 times – once per outgoing edge of w. There are n-1 iterations, which seems to lead to a quadratic number of heap operations.

But, each edge (v,w) of the graph makes at most one appearance in line 12 – when v is first extracted from the heap and moved from V-X to X. Thus, lines 13 and 15 are each performed at most once per edge, for a total of 2m operations, where m is the number of edges.

This heap-based implementation of Dijkstra uses O(m+n) heap operations, each taking O(logn) time. The overall running time is O((m+n)logn).