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.

2. 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)

3. 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 V^2 – ne te kundert nuk jep rezultat te sakte)

4. 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.

5. 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]|.

6. 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.

7. 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**)

8. 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, pra do ndryshonin vlerat dhe do futeshin ne queue vazhdimisht)

9. 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.
   Solution. Per te treguar qe ka nje cikel negative me ane te algoritmit Bellman-Ford, mjafton te shohim nese na duhet te bejme relaksime te tjera pas V-1 relaksimeve total. Nese po, atehere ka cikel negative. Ne momentin qe eshte arritur ne iteracionin e V-te, e kemi te detektuar ciklin negative.
   Implementimi mund te jete i tille: Krijojme nje graf me te gjitha nyjet dhe te gjitha lidhjet e vektorit edgeTo[] pas gjitha iteracioneve te nevojshme, dhe shohim a ka cikle ne kete graf. Bejme dfs ne graf duke filluar nga nyja burim dhe bejme nje kontroll per nyjet fqinje nese ato jane pjese e stackut se dfs apo jo, se nese jane i bie qe ka cikel. Me pas futen ne nje stack te gjitha lidhjet qe jane pjese e ciklit deri tek nyja qe shkaktoi ciklin per te detektuar nengrafin ciklik.

10. **To do**. 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.

11. **To do**. 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.

12. **To do**.Shortest paths in undirected graphs. Write a program DijkstraUndirectedSP.java that solves the single-source shortest paths problems in undirected graphs with nonnegative weights using Dijkstra's algorithm.

13. 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.

    Solution. Pasi te jete bere Dijkstra per te gjitha nyjet si zakonisht, do marrim vleren maksimale te vektorit distTo.

14. 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.

15. 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).
    Solution. Konstruktori i ri qe do kete parameter vetem grafin dhe te gjitha nyjet do kene distancen 0.0 dmth qe algoritmi do ndryshoje ne menyren si funksionon dhe do supozoje qe te gjitha nyjet jane ne distance te njejte nga burimi ne fillim te iteracioneve. Pra, i bie qe kemi nje burim virtual per nyjet. Megjithate, algoritmi vazhdon te relaksoje lidhjet dhe me kalimin e kohes mund te detektoje cikle negative. (me te njejten logjike, dmth nqs pas V-1 kalimeve kemi ndryshime dhe duhet te behet nje relaksim i ri, dmth ka cikel negativ)

Quiz 9.2

Which of the following running times best describes a straightforward implementation of Dijkstra's algorithm for graphs in adjacency-list representation? As usual, n and m denote the number of vertices and edges, respectively, of the input graph.

a) $O(m+n)$

b) $O(m \log n)$

c) $O(n^2)$

d) $O(mn)$

(See below for the solution and discussion.)

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).

Quiz 10.2

How many times does `Dijkstra` execute lines 13 and 15? Select the smallest bound that applies. (As usual, n and m denote the number of vertices and edges, respectively.)

a) $O(n)$

b) $O(m)$

c) $O(n^2)$

d) $O(mn)$

(See below for the solution and discussion.)

Correct answer: (b).

`Dijkstra` (Heap-Based, Part 2)

```
        // update heap to maintain invariant
12      for every edge (w*, y) do
13          DELETE y from H
14          key(y) := min{key(y), len(w*) + ℓ_{w*y}}
15          INSERT y into H
```

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).

Prim, Kruskal

1. Given an MST for an edge-weighted graph G, suppose that an edge in G that does not disconnect G is deleted. Describe how to find an MST of the new graph in time proportional to E.
   Solution. If the edge is not in the MST, then the old MST is an MST of the updated graph. Otherwise, deleting the edge from the MST leaves two connected components. Add the minimum weight edge with one vertex in each component.
   When an edge that is not in the MST of an edge-weighted graph G is deleted, the MST remains unchanged. However, if an edge in the MST is deleted, we need to recompute the MST efficiently. The MST is a tree, so removing an edge disconnects the tree into two disjoint subtrees. Since we removed an edge, we need to find the minimum-weight edge that reconnect the two disconnected components. The new MST can be found in O(E) time just by finding the MST all over again like usual.

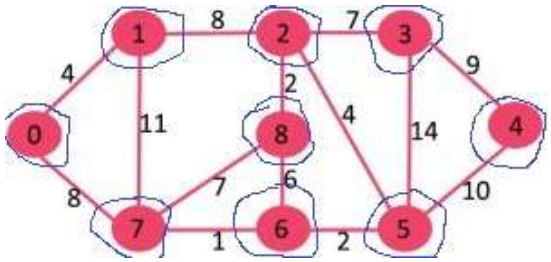
2. Given an MST for an edge-weighted graph G and a new edge e, describe how to find an MST of the new graph in time proportional to V.
   Solution. Add edge e to the MST creates a unique cycle. Delete the maximum weight edge on this cycle. Identify the cycle which consists of only MST edges + the new edge. The MST must have the minimum total weight, so removing the heaviest edge from the cycle restores the MST property. This can be done in O(V) time by traversing the cycle and keeping track of the maximum weight edge.
   If the new edge connects two previously disconnected components, simply add it to the MST.

3. Boruvka's algorithm. Develop an implementation BoruvkaMST.java of Boruvka's algorithm: Build an MST by adding edges to a growing forest of trees, as in Kruskal's algorithm, but in stages. At each stage, find the minimum-weight edge that connects each tree to a different one, then add all such edges to the MST. Assume that the edge weights are all different, to avoid cycles. Hint: Maintain in a vertex-indexed array to identify the edge that connects each component to its nearest neighbor, and use the union-find data structure.
   Remark. There are a most log V phases since number of trees decreases by at least a factor of 2 in each phase. Attractive because it is efficient and can be run in parallel.
   Solution. The complete algorithm:
   1. Input is a connected, weighted and un-directed graph.
   2. Initialize all vertices as individual components (or sets).
   3. Initialize MST as empty.
   4. While there are more than one components, do following for each component:
      - Find the closest weight edge that connects this component to any other component.

- Add this closest edge to MST if not already added.

5. Return MST.

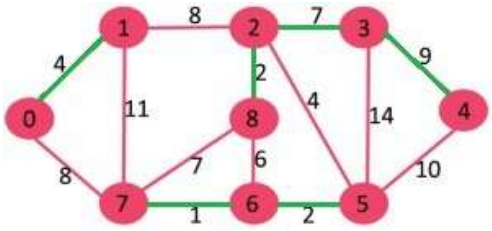
Algorithm example: Initially, MST is empty. Every vertex is single component.



For every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

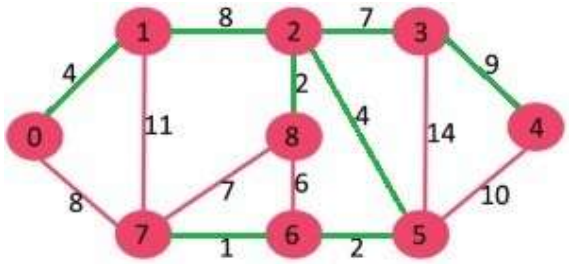
Now MST becomes:



We again repeat the before step for every component, find the cheapest edge that connects it to some other component.

Component	Cheapest Edge that connects it to some other component
{0,1}	1-2 (or 0-7)
{2,3,4,8}	2-5
{5,6,7}	2-5

After this stage, there is only one component which has all edges. Since there is only one component left, we stop and return MST.



4. Minimum bottleneck spanning tree. A minimum bottleneck spanning tree of an edge-weighted graph G is a spanning tree of G such that minimizes the maximum weight of any edge in the spanning tree. Design an algorithm to find a minimum bottleneck spanning tree.
   Solution. Every MST is a minimum bottleneck spanning tree (but not necessarily the converse). This can be proved using the cut property.

5. Minimum median spanning tree. A minimum median spanning tree of an edge-weighted graph G is a spanning tree of G such that minimizes the median of its weights. Design an efficient algorithm to find a minimum median spanning tree.
   Solution. Every MST is a minimum median spanning tree (but not necessarily the converse).

6. Minimum variance spanning tree. Given a connected edge weighted graph, find a minimum spanning tree that minimizes the variance of its edge weights.

7. Suppose that you implement an eager version of Prim's algorithm but instead of using a priority queue to find the next vertex to add to the tree, you scan through all V entries in the distTo[] array to find the non-tree vertex with the smallest value. What would be the order of growth of the worst-case running time for the eager version of Prim's algorithm on graphs with V vertices and E edges? When would this method be appropriate, if ever? Defend your answer.

   Solution. Prim's algorithm would run in time proportional to V^2, which is optimal for dense graphs. Per cdo nyje do kontrollonim te gjitha nyjet.

Undirected Graphs

Quiz 7.1

Consider an undirected graph with n vertices and no parallel edges. Assume that the graph is connected, meaning "in one piece." What are the minimum and maximum numbers of edges, respectively, that the graph could have?

[3]For a finite set S, $|S|$ denotes the number of elements in S.

Measuring the Size of a Graph

a) $n-1$ and $\frac{n(n-1)}{2}$

b) $n-1$ and n^2

c) n and 2^n

d) n and n^n

Correct answer: (a). In a connected undirected graph with n vertices and no parallel edges, the number m of edges if at least n-1 and at most n(n-1)/2. To see why the lower bound is correct, consider a graph G = (V,E). Initially, before any edges are added, each of the n vertices is completely isolated, so the graph trivially has n "distinct" pieces. Adding an edge (v,w) has the effect of fusing the piece containing v with the piece containing w. Thus, each edge addition decreases the number of pieces by at most 1. To get down to a single piece from n pieces, you need to add at least n-1 edges. The maximum number of edges in a graph with no parallel edges is achieved by the complete graph, with every possible edge present. Because there are n(n-1)/2 pairs of vertices in an n-vertex graph, this is also the maximum number of edges.

DFS, BFS

1. Parallel edge detection. Devise a linear-time algorithm to count the parallel edges in a graph.
   Hint: maintain a Boolean array of the neighbors of a vertex, and reuse this array by only reinitializing the entries as needed.

2. Two-edge connectivity. A bridge in a graph is an edge that, if removed, would separate a connected graph into two disjoint subgraphs. A graph that has no bridges is said to be two-edge connected. Develop a DFS-based data type Bridge.java for determining whether a given graph is edge connected.

3. Suppose you use a stack instead of a queue when running breadth-first search. Does it still compute shortest paths?

   Solution. If G is a tree, replacing the queue of the breadth-first search algorithm with a stack will yield a depth-first search algorithm. For general graphs, replacing the stack of the iterative depth-first search implementation with a queue would also produce a breadth-first search algorithm, although a somewhat nonstandard one. => nuk na jep rrugen me te shkurter.

4. DFS with an explicit stack. Give an example of possibility of stack overflow with DFS using the function call stack, e.g., line graph. Modify DepthFirstPaths.java, so that it uses an explicit stack instead of the function call stack.

5. **Nonrecursive depth-first search**. Write a program NonrecursiveDFS.java that implements depth-first search with an explicit stack instead of recursion.

6. Nonrecursive dfs. Exaplain why the following nonrecursive method (analogous to BFS but using a stack instead of a queue) does not implement dfs.

```
private void dfs(Graph G, int s) {
    Stack<Integer> stack = new Stack<Integer>();
    stack.push(s);
    marked[s] = true;
    while (!stack.isEmpty()) {
       int v = stack.pop();
       for (int w : G.adj(v)) {
           if (!marked[w]) {
               stack.push(w);
               marked[w] = true;
               edgeTo[w] = v;
           }
       }
    }
}
```

Solution: Consider the graph consisting of the edges 0-1, 0-2, 1-2, and 2-1 with vertex 0 as the source.
The code is not a strict dfs because it processes all neighbors of a node before moving deeper; the traversal order is affected by the adjacency list iteration, making it closer to bfs with a stack rather than a true dfs. By adjusting the neighbor processing order, we can ensure a proper dfs traversal.

7. Delete a vertex without disconnecting a graph. Given a connected graph, design a linear-time algorithm to find a vertex whose removal (deleting the vertex and all incident edges) does not disconnect the graph.
   Hint 1 (using DFS): run DFS from some vertex s and consider the first vertex in DFS that finishes.
   Hint 2 (using BFS): run BFS from some vertex s and consider any vertex with the highest distance.

8. All paths in a graph. Write a program AllPaths.java that enumerates all simple paths in a graph between two specified vertices.
   Hint: use DFS and backtracking. Warning: there many be exponentially many simple paths in a graph, so no algorithm can run efficiently for large graphs.

Topological order, Kosaraju-Sharir

1. What are the strong components of a DAG?
   Solution: Each vertex is its own strong component.

2. True or false: The reverse postorder of a digraph's reverse is the same as the postorder of the digraph.
   Solution: False.

3. True or false: If we modify the Kosaraju-Sharir algorithm to run the first depth-first search in the digraph G (instead of the reverse digraph G^R) and the second depth-first search in G^R (instead of G), then it will still find the strong components.
   Solution. True, the strong components of a digraph are the same as the strong components of its reverse.

4. True or false: If we modify the Kosaraju-Sharir algorithm to replace the second depth-first search with breadth-first search, then it will still find the strong components.
   Solution. True.

5. Directed Eulerian cycle. A directed Eulerian cycle is a directed cycle that contains each edge exactly once. Write a digraph client DirectedEulerianCycle.java that find a directed Eulerian cycle or reports that no such cycle exists.
   Hint: Prove that a digraph G has a directed Eulerian cycle if and only if vertex in G has its indegree equal to its outdegree and all vertices with nonzero degree belong to the same strong component. (without this check, dfs might return a path instead of a cycle)
   Zbatojme dfs me kushtin qe nr i indegree te jete i barabarte me nr e outdegree dhe ne nje stack tjeter vendosim lidhjet qe do mbajne ciklin.

6. **Todo**. Strong component. Describe a linear-time algorithm for computing the strong component containing a given vertex v. On the basis of that algorithm, describe a simple quadratic-time algorithm for computing the strong components of a digraph.
   Partial solution: To compute the strong component containing s
   Find the set of vertices reachable from s
   Find the set of vertices that can reach s
   Take the intersection of the two sets
   Using this as a subroutine, you can find all strong components in time proportional to t (E + V), where t is the number of strong components.

7. Hamiltonian path in DAGs. Given a DAG, design a linear-time algorithm to determine whether there is a directed path that visits each vertex exactly once.
   Solution: Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order. Nese ka topological order atehere i bie qe ka dhe Hamiltonian path.

8. Queue-based topological order algorithm. Develop a nonrecursive topological sort implementation TopologicalX.java that maintains a vertex-indexed array that keeps track of the indegree of each vertex.
   Solution.Initialize the array and a queue of sources (queue of sources dmth ato qe kane indegree 0) in a single pass through all the edges. Then, perform the following operations until the source queue is empty:
   Remove a source from the queue and label it.
   Decrement the entries in the indegree array corresponding to the destination vertex of each of the removed vertex's edges.
   If decrementing any entry causes it to become 0, insert the corresponding vertex onto the source queue.

9. **To do**.Shortest directed cycle. Given a digraph, design an algorithm to find a directed cycle with the minimum number of edges (or report that the graph is acyclic). The running time of your algorithm should be proportional to E V in the worst case.
   Application: give a set of patients in need of kidney transplants, where each patient has a family member willing to donate a kidney, but of the wrong type. Willing to donate to another person provided their family member gets a kidney. Then hospital performs a "domino surgery" where all transplants are done simultaneously.

   Solution: run BFS from each vertex s. The shortest cycle through s is an edge v->s, plus a shortest path from s to v. ShortestDirectedCycle.java