Example sheet 5

Graph traversal, path finding
Algorithms—DJW*–2018/2019

Questions labelled ∘ are warmup questions. Questions labelled ∗ involve more thinking and you are not expected to tackle them all.

Question 1°. Read the definitions of *directed acyclic graph* and *tree* from lecture notes. Draw an example of each of the following, or explain why no example exists:
(i) A directed acyclic graph with 8 vertices and 10 edges
(ii) An undirected tree with 8 vertices and 10 edges
(iii) A graph without cycles that is *not* a tree

Question 2°. Show that `dfs_recurse` has running time $O(V + E)$.

Question 3. The algorithms `dfs` and `dfs_recurse` (as given in the handout) do not always visit vertices in the same order. Modify `dfs` so that they do. Give pseudocode. *[Assume that there is an ordering on vertices, and that* v`.neighbours` *returns a sorted list of* v*'s neighbouring vertices.]*

Question 4. Give pseudocode for a function `dfs_recurse_path(`g, s, t`)` based on `dfs_recurse`, that returns a path from s to t. Modify your function so that it does not visit any further vertices once it has reached the destination vertex t.

Question 5. Consider a directed graph in which every vertex v is labelled with a real number x_v. For each vertex v, define m_v to be the minimum value of x_u among all vertices u such that either $u = v$ or u is reachable via some path from v. Give an $O(E + V \log V)$-time algorithm that computes m_v for all vertices v.

Question 6. Modify `bfs_path(`g, s, t`)` to find *all* shortest paths from s to t.

Question 7 (FS)°. In a directed graph with edge weights, give a formal proof of the triangle inequality

$$d(u, v) \leq d(u, w) + c(w \to v) \quad \text{for all vertices } u, v, w \text{ with } w \to v$$

where $d(u, v)$ is the minimum weight of all paths from u to v (or ∞ if there are no such paths) and $c(w \to v)$ is the weight of edge $w \to v$. Make sure your proof covers the cases where no path exists.

Question 8. There is a missing step in the proof of Dijkstra's algorithm, as given in the handout. The proof looks at the state of program execution at the point in time at which some vertex v fails the assertion on line 9. Call this time t. It uses the equation

$$u_{i-1}\texttt{.distance} = \text{distance}(s \text{ to } u_{i-1}),$$

and justifies it by saying "the assertion on line 9 didn't fail when u_{i-1} was popped." Let t' be the time when u_{i-1} was popped, and explain carefully why u_{i-1}`.distance` cannot change between t' and t.

Question 9 (CLRS-24.3-4). A contractor has written a program that she claims solves the shortest path problem, on directed graphs with edge weights ≥ 0. The program produces v`.distance` and v`.come_from` for every vertex v in a graph. Give an $O(V + E)$-time algorithm to check the output of the contractor's program. *[Hint. Pay attention to the case where some edge weights are zero.]*

*Questions labelled FS are from Dr Stajano. Questions labelled CLRS are from *Introduction to Algorithms, 3rd ed.* by Cormen, Leiserson, Rivest and Stein.

Question 10. Suppose we have implemented Dijkstra's algorithm using a priority queue for which `push` and `decreaskey` have running time $\Theta(1)$, and `popmin` has running time $\Theta(\log n)$ where n is the number of items in the queue. By constructing a suitable family of graphs, show that Dijsktra's algorithm has worst-case running time $\Omega(E + V \log V)$. You may assume $|E| \geq |V| - 1$.

Question 11 (FS). In Section 5.5, we defined $M_{ij}^{(\ell)}$ to be the minimum weight among all paths from i to j that have ℓ or fewer edges, and we derived the recursion

$$M^{(1)} = W, \qquad M^{(\ell)} = W \otimes M^{(\ell-1)} \tag{1}$$

where $W_{ij} = 0$ if $i = j$, $W_{ij} = \text{weight}(i \to j)$ if there is an edge $i \to j$, and $W_{ij} = \infty$ otherwise. How can we define $M^{(0)}$ so that $M^{(1)} = W \otimes M^{(0)}$? *[Hint. For normal matrix multiplication, if A is any matrix and I is the identity matrix then $A = A\,I$.]*

Question 12. Let $P_{ij}^{(\ell)}$ be the pair (d, p) where d is the minimum weight among all paths from i to j that have ℓ or fewer edges, and p is one such path. Define a recursion for $P^{(\ell)}$ similar to equation (1), and explain the operator you have invented to replace $\otimes$.

Question 13°. By hand, run both Dijkstra's algorithm and the Bellman-Ford algorithm on each of the graphs below, starting from the shaded vertex. The labels indicate edge costs, and one is negative. Does Dijkstra's algorithm correctly compute minimum weights?

[Run `dijkstra` as described in lectures, which loops until `toexplore` is empty. Some textbooks use different versions of Dijkstra's algorithm, that terminate once a destination vertex has been popped, or that don't allow popped vertices to re-enter `toexplore`.]

Question 14. In the course of running the Bellman-Ford algorithm, is the following assertion true? "Pick some vertex v, and consider the first time at which the algorithm reaches line 7 with v`.minweight` correct i.e. equal to the true minimum weight from the start vertex to v. After one pass of relaxing all the edges, u`.minweight` is correct for all $u \in \text{neighbours}(v)$."

If it is true, prove it. If not, provide a counterexample.

Question 15 (CLRS-25.3-4). An engineer friend tells you there is a simpler way to reweight edges than the method used in Johnson's algorithm. Let w^* be the minimum weight of all edges in the graph, and just define $w'(u \to v) = w(u \to v) - w^*$ for all edges $u \to v$. What is wrong with your friend's idea?

Question 16*. You've learned what $O(n)$ means when there's a single variable n that increases. How would you define $O(E + V \log V)$?

Question 17*. Give a $O(V + E)$ algorithm that solves Question 5. *[This goes beyond what is taught in lectures. Credit to Georgiev (matriculated 2016) for spotting this.]*

Question 18*. Consider a graph without edge weights, and write $d(u, v)$ for the length of the shortest path from u to v. The *diameter* of the graph is defined to be $\max_{u,v \in V} d(u, v)$. Give an efficient algorithm to compute the diameter of an undirected tree, and analyse its running time. *[Hint. Use breadth-first search, twice.]*

Question 19*. The Bellman-Ford code given in the handout will report "Negative cycle detected" if there is a negative-weight cycle reachable from the start vertex. Modify the code so that, in such cases, it returns a negative-weight cycle, rather than just reporting that one exists.

Question 20. (Not intended for supervision.) Here is a question about Dijsktra's algorithm, followed by a selection of actual answers written by students. Read these answers and pinpoint carefully where they are lacking.

> Let `dijkstra_path`(g, s, t) be an implementation of Dijkstra's shortest path algorithm that returns the shortest path from vertex s to vertex t in a graph g. Prove that the implementation can safely terminate when it first encounters vertex t.

[The underlying issue is that these are word salad answers, not maths, and it's easy to slip wishful thinking into a word salad. That is, these answers assume something is true but they don't prove it or even state it. It's easy to fall into the trap of wishful thinking. You may have fallen into it yourself when answering question 14.

[The most common wishful thinking here is to take for granted that v `.distance` *as computed by the algorithm at a certain point in time, is equal to the true shortest path distance from s to v. Do the answers assume this is true for all vertices in the frontier? For all vertices popped before t? Do they prove it?*

[Dijsktra's algorithm requires edge weights to be ≥ 0. If some edges have negative weights then the algorithm might not terminate—and so its proof of correctness must break! Any "proof" that doesn't use the assumption must be wrong. Where do these answers use the assumption?]

Bad answer 1. Proof by induction. For the base case, consider a graph with one edge and two vertices. There is only one path, so when it reaches t that path is the shortest path and the algorithm can terminate.

For the induction step, assume that when the nth vertex is popped it has the correct distance. The $(n+1)$th vertex reached, call it v, is then the next closest (to s) adjacent vertex. If a closer vertex existed, it would have to go through a path of a shorter distance, so we would have reached that vertex before v on this path. But in this case Dijkstra would already have reached it. This can't happen, so no shorter path to it exists.

Eventually, we will reach t, and by the above proof there is no shorter path to t, so `dijkstra_path` can safely terminate.

Bad answer 2. Assume that "first encounters" means that t has just been popped from the priority queue. Let S be the set of all vertices that have been popped. (Initially, $S = \{s\}$, and s`.distance` $= 0$.) Let the true shortest path be

$$s = v_1 \to v_2 \to \cdots \to v_k = t.$$

Suppose t has just been popped, and let the edge that gives it its distance be $x \to t$ where $x \in S$. Consider any other path of the form

$$s = v_1 \to \cdots \to x \to y \to \cdots \to t$$

where $y \notin S$. Since t has just been picked and y has not, we know that the priority key for t is $\leq$ that for y, therefore the distance to t on the first path is smaller than that on the other path. Therefore the path chosen is the true shortest path.

Bad answer 3. Dijkstra's algorithm performs a depth-first search on the graph, storing a frontier of all unexplored vertices that are neighbours to explored vertices.

Each time it chooses a new vertex v to explore, from the frontier of unexplored vertices, it chooses the one that will have the shortest distance from the start s, based on the edge weight plus the distance from its already explored neighbour.

Given that no other vertex in the frontier is closer to s, and that this new vertex v has yet to be explored, when v is explored it must have been via the shortest path from s to v.

Hence, when t is first encountered, it must have been found via its shortest path and the program can safely terminate.

Bad answer 4. At the moment when the vertex t is popped from the priority queue, it has to be the vertex in the priority queue with the least distance from s. This means that any other vertex in the priority queue has distance $\geq$ that for t. Since all edge weights in the graph are ≥ 0, any path from s to t via anything still in the priority queue will have distance $\geq$ that of the distance from s to t when t is popped, thus the distance to t is correct when t is popped.

Bad answer 5. In Dijkstra's algorithm we add vertices to a min priority queue sorted by the shortest distance to them from the start vertex s. We then repeatedly pop the minimum element in the priority queue.

Assume that at the time we pop t, there is a shorter path to it than the one we just found. If that is the case, then there is some vertex x that immediately precedes t in the shorter path.

However, this x cannot exist, as if it did it would have been popped from the frontier before t (path $s \rightsquigarrow x$ must be shorter than the path $s \rightsquigarrow t$) and is already part of the path being considered. We reach a contradiction, so the assumption that there is a shorter path than the one found must be false.

Bad answer 6. Assume that upon termination the path to t is suboptimal. Let v be the vertex which is first in the computed path that is not on the shortest path. Let u_i be the node on the shortest path that should have been chosen instead of v.

$$s = u_1 \to u_2 \to \cdots \to u_{i-1} \to v \to \cdots \to t \quad \text{(computed path)}$$
$$s = u_1 \to u_2 \to \cdots \to u_{i-1} \to u_i \to \cdots \to t \quad \text{(true shortest path)}$$

Since v is not optimal,

$$\begin{aligned}
&\text{distance}(s \text{ to } v) < v\texttt{.distance} && \text{since } v \text{ is not optimal} \\
&\leq \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to v) && \text{as it didn't fail at } u_{i-1} \\
&\leq \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i) && \text{as } v \text{ was chosen over } u_i \text{ in priority queue} \\
&\leq \text{distance}(s \text{ to } v) && \text{as } u_i \text{ is on the shortest path to } v.
\end{aligned}$$

We have proved that distance(s to v) < distance(s to v), a contradiction, hence the path must be the shortest.