Algorithms and Data Structures (WS15/16)

Example Solutions for Units 21 & 22

Problem 1

We find the longest path in a DAG with unweighted edges (so all edges have weight 1).

We saw in Unit 20 Problem 2 that any DAG has a topological ordering and we can compute it in $\mathcal{O}(n+m)$ using DFS.

Let d[v] be the length of the longest path ending in vertex v. We can compute d by processing the vertices of the graph in reverse topological order. Then, when we reach a node v, all the incoming edges come from nodes higher in the topological ordering. For these we have already computed the longest path. We pick the maximum of these and add 1 to get the longest path ending in v.

Pseudocode:

```
1: procedure LengthLongestPath(G(V, E))
2:     Compute Topological Ordering for G
3:     for v ∈ V in reverse topological order do
4:         m ← 0
5:         for w such that (w, v) ∈ E do
6:             if d[w] > m then
7:                 m ← d[w]
8:         d[v] ← m + 1
```

Note that since we look at incoming edges, we need to preprocess the graph and store these for each vertex. This will take time $\mathcal{O}(n+m)$. Since we only look at each node and edge once, the running time of the procedure is $\mathcal{O}(n+m)$.

Now we need to recover the longest path from d. First we find the node v with the maximum d[v] value. The longest path will end in v. We reconstruct the longest path by looking at all the incoming edges (w, v) and taking the one with the largest d-value. We stop when we reach a node w with d[w] = 1.

This procedure again takes $\mathcal{O}(n+m)$, which gives us a total running time of $\mathcal{O}(n+m)$.

```
1: procedure LongestPath(G(V, E), d)
2:     Find v with maximum d[v] value
3:     while d[v] > 1 do
4:         for w such that (w, v) ∈ E do
5:             if d[w] = d[v] − 1 then
6:                 add (w, v) to the beginning of the longest path
7:                 v ← w
8:                 break out of for loop
```

Problem 2

Note about running time: all of the solutions in the exercises take $\mathcal{O}(m+n)$ to prepare, and then run the Bellman-Ford algorithm, which takes $\mathcal{O}(mn)$. Therefore, any of these can be run efficiently in $\mathcal{O}(mn)$.

a) We set the weight on red edges to -1 and on blue edges to 2. Then we use the Bellman-Ford algorithm to get the shortest path. We observe that the total weight of the shortest path is of the form $-1r+2b$, where r and b are the number of red and blue edges, respectively. If the shortest path has non- positive weight, that means that $-1r+2b \leq 0$, or equivalently, $r \geq 2b$, that is, the number of red edges is at least twice the number of blue edges. On the other hand, if the shortest path has positive weight, then there is no path such that $-1r+2b \leq 0$ holds, and therefore, no path for which the number of red edges is at least twice the number of blue edges.

Let us now consider what happens if the Bellman-Ford algorithm reports that there is a cycle. If $d[t] \leq 0$, that is, the distance computed from s to t is non-positive, we don't mind negative cycles existing, because the argument above applies. However, if $d[t] > 0$, we can't be sure if there isn't a path from s to t with a negative cycle, so we need to modify the algorithm slightly. After all edges are relaxed n times, if $d[t] \leq 0$, then we know there is a path as desired, and we can immediately return true. Otherwise, we must modify the negative cycle detection part. In the original, we check if some edges still need relaxing, and if they do, we know that there is a negative cycle.

In the modified version, we run DFS traversing edges backwards, starting from t and find out, for each vertex v, if we can get from v to t. Then, for each negative cycle that is detected as in the original algorithm, we check if we can get from this vertex to t, by using the results previously computed using DFS. If we can, then we know that there is a path from s to t that has a negative cycle, and therefore, by going over the cycle enough times, we get $-1r+2b \leq 0$.

b) We just set the weight on red edges to 1 and on blue edges to 0. Then, the total weight of a path is given by the number of red edges in the path, and running the Bellman-Ford algorithm finds the path for which this quantity is minimized.

c) For this problem, we will build an instance based on the input graph, and run a shortest path algorithm (or pretty much any algorithm that decides if a path exists between two nodes) on it to find the desired path exists. For convenience, we denote by s and t the source and destination of our path (u and v in the problem statement).

Given an input graph G, we make three copies of the graph, which we call G_0, G_1, G_2. The index in each graph represents the number of red edges travelled so far. Now, we join the three copies into one graph as follows. Red edges (u_0, v_0) in G_0 are changed to (u_0, v_1), that is, we connect u_0 to v_1, which is the same vertex as v_0, but on graph G_1, and remove the original

connection. Similarly, a red edge (u_1, v_1) in G_1 becomes now (u_1, v_2), where v_2 is the vertex corresponding to v_1 in G_2. All red edges in G_2 are deleted.

If there is a path between s_0 (vertex s in G_0) and t_2 (vertex t in G_2), then there must be a path traversing exactly two red edges, since a path between s_0 and t_2 crosses exactly two red edges: one when the path goes from G_0 to G_1, and the other when it goes from G_1 to G_2. Since all red edges in G_2 were deleted, the path must cross exactly two edges. Conversely, if there is a path that crosses exactly two red edges in the original graph, then we can represent this path in the new instance, where the two red edges cross into G_1 and then G_2. Therefore, the two representations are equivalent.

Problem 3

a) The SSSP will not terminate if the input graph contains a negative cycle. After k iteration we will have relaxed all edges on any path of length k in the graph. Thereby the algorithm will have computed all shortest path with at most k edges. As a shortest path can contain at most $n-1$ edges we know that any improvement after the nth iteration indicates the existence of a negative cycle. Therefore we replace the outer while loop with an for loop that looks like that.

   for i=1 to n and Improved $\neq \emptyset$ do

   No other changes are necessary.

b) Consider a direct path on n vertices. Bellmann-Ford will take $O(mn)$ time but the SSSP only $O(n)$ time the distance of every node to v_1 will be improved exactly once.

c) Consider a graph that consists of one cycle where all edges have negative weights and an additional vertex s that has an outgoing edge to all vertices in the cycle. We want to solve SSSP for s. After the first iteration all vertices of the cycle will be in the improved set. None of them will leave the set and in every iteration the algorithm will relax all edges in the cycle. Therefore the algorithm will take $O(nm)$ time.

Problem 4

a) We can simply modify Dijkstra's algorithm:

   - instead of the distance to a node, we will look at the width of a path to a node; we want to find the widest possible path
   - we initialize the $widthto$(Voelklingen) $= \infty$ and $withto(v) = 0$ for all other nodes
   - from the set of unexplored nodes we pick the one which has the maximal $widthto$ value
   - update: for every neighbor v of a node u, we compute $new = \min\{widthto(u), width(u, v)\}$ and if this is larger than $widthto(v)$, we set $widthto(v) = new$

b) The Bellman-Ford algorithm can still be applied if we assume a total ordering (means that every two elements are comparable; if we do not assume this - we might encounter situations in which we can not decide which path is longer...):

- we initialize the weight of the source to be equal to the neutral element, all others to infinity (where infinity is a value greater than every element of the semi-group)
- relaxation of edges is still performed correctly: we replace $+$ by $\bigoplus$ and elements can be compared because there is a total ordering ($x < y \iff (x \leq y \wedge x \neq y)$)
- the negative-weight cycle check still works

c)
- Dijkstra's algorithm can be applied under the condition that there are no "negative" elements in the ordered semigroup (actually just on the graph edges), where negative means elements a such that $a < 0$