CS310: Advanced Data Structures and Algorithms
Fall 2018 Assignment 4
Due: Monday, November 19 2018, in class

Goals
Greedy algorithms (graphs and others), Dynamic Programming

Reading
K&T, Chapter 4 (greedy algorithms), Chapter 6 (dynamic programming) S&W, Chapter 4.3–4.4 (Shortest paths, MSTs)

Questions
1. Solve the midterm exam. If your answer got the full points you can copy it as-is. Otherwise, solve it from scratch. Attach your answers as part of your submission. This means that overall this homework contains 10 questions. A soft copy of the exam is attached as a handout.

2. Given the following weighted, undirected graph:

   ![Graph](image)

   (a) Show the process of Kruskal’s algorithm to find its minimal spanning tree. The format of your answer should be the following: write down the edges in the order in which they are processed, and indicate for each edge whether it appears in the final MST or not.

   (b) Do the same with Prim’s algorithm. Start from $v_1$.

   (c) Draw the final MST (despite possibly selecting the edges in a different order, the MST should be the same for (a) and (b)!).

3. (Adapted from K&T, 4.9): One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible. Specifically, let $G = (V, E)$ be a connected graph with $n$ vertices, $m$ edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of $G$; we define the bottleneck edge of $T$ to be the edge of $T$ with the greatest cost. A spanning tree $T$ of $G$ is a minimum-bottleneck spanning tree (MBST) if there is no spanning tree $T'$ of $G$ with a cheaper bottleneck edge.
(a) Show that every minimum spanning tree of \( G \) is also a minimum-bottleneck tree of \( G \) (it’s easiest to prove by contradiction IMO).

(b) The opposite is not always true. Show an example of a minimum-bottleneck tree of \( G \) which is not a minimum spanning tree of \( G \).

4. (Adapted from K&T, 4.5) Let’s consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint, so each house is a point on an interval) Further, let’s suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations (this actually means that each station covers 8 miles – four to the left, four to the right). Here is a greedy algorithm that achieves this goal, using as few base stations as possible:

- Place the first station 4 miles to the east of the westernmost house.
- Repeat, placing each station 4 miles to the east of the first uncovered house.

Show that this greedy algorithm gives the optimal solution (with the minimum number of stations). You can start by showing that there must be an optimal solution that places the first station 4 miles to the east of the westernmost house. (the reasoning can be similar to what’s discussed in class about the interval scheduling).

5. (Adapted from K&T 6.3) Here is a suggested greedy algorithm to find the longest path in a DAG (directed acyclic graph):

1. Let \( w = v_1 \)
2. Let \( L = 0 \) (the length of the longest path so far)
3. While there is an edge out of \( w \):
   i. Choose an edge \((w, v_j)\) such that \( j \) is minimum (in the example below when \( v_1 \) is considered it would be \( v_2 \), since 2 is the minimum out of \( v_2 \) and \( v_4 \), the outgoing neighbors of \( v_1 \)).
   ii. \( w \leftarrow v_j \)
   iii. \( L \leftarrow L + 1 \)
4. Return \( L \)

(a) Show that the algorithm above indeed gives the longest path in the DAG below:

(b) Slightly modify the graph in (a) such that this algorithm no longer gives the longest path (you may delete or add edges, as long as the graph is still a DAG and still connected). Explain briefly why the algorithm above doesn’t work on your example.

(c) It is possible to find the longest path in a DAG using dynamic programming. This is done by calculating, for every vertex \( v_i \), the longest path that ends in \( v_i \). Notice that any vertex \( v_i \) extends the longest paths ending at each one of its incoming neighbors by 1 (this is true only in DAGs, of course, due to the fact that paths only go one way). Therefore, the length of the longest path ending at \( v_i \) is the length of the longest path of all of \( v_i \)’s predecessors + 1. The algorithm is given below:

   i. Topologically sort the graph
   ii. For each \( v_i \) whose in-degree is 0, set \( LP(v_i) = 0 \) (this is the longest path ending at \( v_i \)).
   iii. For each of the other vertices \( w_i \), in the topologically sorted order:
      
   set \( LP(w_i) = \max_{v_j \in \text{in}(v_i)} \ max_{v_k \in \text{out}(v_j)} LP(v_j) + 1 \)
   iv. Return \( \max_{v_i \in V} LP(v_i) \)
For each vertex in the DAG shown in (a) above, fill the length of the longest path ending in it.

\[
\begin{array}{c|c}
  v_i & LP(v_i) \\
  \hline
  v_1 & \\
  v_2 & \\
  v_3 & \\
  v_4 & \\
  v_5 & \\
\end{array}
\]

(d) What is the run time of the algorithm in (c) above as a function of \(|V|\) (the number of vertices) and/or \(|E|\) (the number of edges)? Explain briefly.

6. (Adapted from K&T, 6.22) To assess how “well-connected” two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the number of shortest paths. This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed weighted graph \(G = (V, E)\). Let us assume all weights are positive. Given the graph and two nodes, \(s\) and \(t\), give an efficient algorithm – a modification of Dijkstra’s, that computes the number of shortest paths in \(G\). The algorithm should not list all the paths; just the number suffices.