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. See solved copy of the midterm
2. Given the following weighted, undirected graph:

(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.

In Kruskal’s algorithm, we greedily select edges in increasing order of weight, as long as they don’t close a cycle.

\[ v_3 - v_4 \]
\[ v_2 - v_3 \]
\[ v_3 - v_4 \] (not a tree edge, closes a cycle)
\[ v_1 - v_4 \]
\[ v_1 - v_5 \]

The other edges are certainly not tree edges, so we can stop here.

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

In Prim’s algorithm, we start from an arbitrary vertex and greedily grow the tree from it, selecting the smallest edge that crosses the cut.
(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).

Let us define an MST $T$, which is not equal to an MBST. In this case, there is a tree whose bottleneck edge $e = (v, w)$ is lighter than the bottleneck edge $e' = (v', w')$ of the MST. Let us define a cut in the graph such that $v$ and $v'$ are on one side and $w$ and $w'$ are on the other (the other vertices can be assigned at random). Both $e$ and $e'$ cross the cut, and obviously $e'$, the edge selected for the MST, is not the lightest edge that crosses the cut ($e$ is lighter than it). But we showed in class that an MST always contains a lightest edge that crosses any cut, contradicting the fact that $T$ is an MST.

(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 $G$.

Here is an example based on the question above. You can start out from the MST, leave the heaviest edge as-is but replacing a lighter edge with a heavier one.

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).

In what follows I look at the houses from left to right, that is – from west to east. So the first house is the westernmost. We argue that there must be an optimal solution that places the first station exactly 4 miles from the first house, which is the greedy selection. Given an optimal solution \( S \), it certainly has a station within 4 miles or less of the first (westernmost) house. This is true for every solution (not necessarily the optimal) since every house needs to be covered. Therefore, exactly 4 miles from the first house is the farthest point you can place the first station. If the first station selected by \( S \) is not at exactly 4 miles from the first house we can ”move” it east so it will be exactly 4 miles from the first house. The move doesn’t spoil the solution, since all the houses covered by the first station are still covered, including the first one. The new placement does not change the number of stations, since we didn’t add any new stations. Therefore, the number of stations is the same as the optimal solution and the greedy selection is safe for an optimal solution.

This logic can be applied to any subsequent station. i.e. – any next station can be placed exactly 4 miles east of the next uncovered house in an optimal solution, since given any optimal solution that doesn’t do it, we can ”move” the station to 4 miles east of the next uncovered house and get an optimal solution with all houses covered.

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

(a) Let \( w = v_1 \)

(b) Let \( L = 0 \) (the length of the longest path so far)

(c) 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. \( L \leftarrow L + 1 \)

(d) Return \( L \)

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

![Diagram]

The path returned by the algorithm is \( v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5 \), which is indeed the longest.

b. Slightly modify the graph in (a) such that this algorithm no longer gives the longest path (Hint: It’s enough to modify two edges: Delete \((v_1, v_2)\) and add another edge, you have to find out where). Explain briefly why the algorithm above won’t work on your example.
The algorithm in (a) will give $v_1 \rightarrow v_4 \rightarrow v_5$, while the longest path is $v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5$. Other examples exist too. I accepted any example that works.

c. It is possible to find the longest path 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, so to speak). 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:

(a) Topologically sort the graph
(b) For each $v_i$ whose in-degree is 0, set $LP(v_i) = 0$ (this is the longest path ending at $v_i$).
(c) For each of the other vertices $w_i$, in the topological sort order:
   i. set $LP(w_i) = \max_{v_j \in \text{predecessors of } w_i} (LP(v_j) + 1)$
(d) Return $\max_{v_i \in V} (LP(v_i))$

For each of the vertices in the DAG shown in (a) above, fill in the length of the longest path ending in it.

<table>
<thead>
<tr>
<th>$v_i$</th>
<th>$LP(v_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>1</td>
</tr>
<tr>
<td>$v_3$</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>2</td>
</tr>
<tr>
<td>$v_5$</td>
<td>3</td>
</tr>
</tbody>
</table>

d. (4%) 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.

$O(|V| + |E|)$ for the topological sorting, and the same for searching the longest path. Notice that the inner loop in the algorithm above depends solely on the number of edges (similar to DFS and BFS shown in class). Hence, the run time is $O(|V| + |E|)$.

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.

The idea is to keep track of the number of paths as you relax edges. We add a variable to each vertex $v$, $paths[v]$. The boundary condition is that $paths[s] = 1$ (there is only one path – an empty one...). Otherwise, when you relax an edge $e = v \rightarrow w$, one of three things may happen:

- $dist[v] + e > dist[w]$. In this case do nothing, since the path through $v$ is not a shortest path.
- $dist[v] + e < dist[w]$. In this case we found a strictly shorter path to $w$. This path goes through $v$, so all the shortest paths to $v$ are also shortest paths to $w$. Set $paths[w] = paths[v]$.
- $dist[v] + e == dist[w]$. In this case we found an equally short path to the shortest ones known for $w$. We have to add the new shortest paths to the ones we already know. Set $paths[w] += paths[v]$. 