Recap of this week’s lectures:

- Dynamic programming
- Single-source shortest path algorithms: Dijkstra and Bellman-Ford
- All-pairs shortest path algorithms: Floyd-Warshall and Matrix-Mult
- Subset DP for TSP

1. **Bottleneck Paths.** Given a directed graph $G$, suppose each edge has a non-negative capacity $c_e$. Given a directed path from $s$ to $t$, the bottleneck edge of this path is the min-capacity edge on it. The $s$-$t$ bottleneck path asks for such a path whose bottleneck edge capacity is as large as possible.

Show how to modify Dijkstra’s algorithm to solve this problem in $O(m \log n)$ time (or $O(m + n \log n)$ time using Fibonacci heaps).

**Solution:** The basic idea is as follows: Start from the set $X = s$, and find the vertex adjacent to $X$ that has the highest bottleneck. Add that vertex to $X$.

**Claim** (Dijkstra’s Principle for bottlenecks): For any partition of $V$ into $X$ with $s \in X$ and $Y = V \setminus X$, let

$$p(v) = \max_{x \in X} (\min(bottle(s, x), cap(x, v)))$$

Then $\max_{y \in Y} p(y) = \max_{y \in Y} bottle(s, y)$

**Proof.** Let a vertex $v \in Y$ be such that $bottle(s, v) = \max_{y \in Y} bottle(s, y)$, and consider the path from $s$ to $v$. Since the path starts at $s \in X$, there must be some $u \in Y$ such that the path goes from a vertex in $X$ to $u$. However, the bottleneck is the minimum of all capacities along the path, so $bottle(s, u) \geq bottle(s, v)$, which means that $\max_{y \in Y} p(y) \geq bottle(s, u) \geq bottle(s, v) \geq \max_{y \in Y} bottle(s, y)$, which completes the proof.

So now we do the same as Dijkstra’s Algorithm, keeping a max heap of values adjacent to our current set, and adding the max to our set. To update the heap we insert the minimum of the bottleneck and the capacity to the new vertex.

2. **Johnson’s Algorithm.** We did not get a chance to discuss Johnson’s algorithm for APSP, we do that now. (Details in notes, Section 4.3 of Lecture 8).
(a) If the edge-weights are non-negative, running Dijkstra from each node takes $n \cdot O(m + n \log n)$ time. Much faster than F-W if the graph is sparse, i.e., $m \ll n^2$.

**Solution:** Seems good

(b) Suppose we assign an “offset” $\Phi_v$ to each vertex $v$, and define the offset edge-length of edge $(u, v)$ to be $\ell'_{uv} := \Phi_u + \ell_{uv} - \Phi_v$. Show that a $s$-$t$ path is a shortest path with respect to lengths $\ell$ if and only if it is a shortest path with respect to lengths $\ell'$.

**Solution:** Consider any path from $s$ to $t$, and let's say that it goes through a vertex $v$. Then there is an edge into $v$ (that contributes a $\Phi_v$), and an edge out of $v$, which contributes a $-\Phi_v$. So when we sum all of the edges on the path, any internal vertices cancel out, leaving us with $\Sigma \ell_{uv} + \Phi_s - \Phi_t$. This holds for any path, and since $\Phi_s$ and $\Phi_t$ are constant, the shortest path with the new edges weights is the same.

(c) Call an offset “feasible” if $\ell'_{u,v} \geq 0$ for all edges, even if $\ell$ could have been negative. Suppose there is a vertex $x$ that has finite distance to every other vertex. Show that $\Phi_v := dist(x, v)$ is a feasible offset.

**Solution:** We want to show that $\ell'_{u,v} \geq 0$, or equivalently, that $\ell_{uv} \geq \Phi_v - \Phi_u$. We can replace the potentials with the distances to $x$, giving us the following: $\ell_{uv} \geq dist(x, v) - dist(x, u)$ however the distance from $x$ to $v$ is upperbounded by the distance from $x$ to $u$ plus the edge length from $x$ to $v$, because this is a valid path from $x$ to $v$, which tells us that $dist(x, v) \leq dist(x, u) + \ell_{uv}$. Rearranging finishes the problem.

(d) Infer that you can compute APSP by running Bellman-Ford once (to compute the feasible offset) and then $n$ Dijkstras.

**Solution:**

i. Create a dummy node and connect it to every node in the graph with length 0.

ii. Run Bellman-Ford from the dummy node and get distances to every vertex in the graph. If you see a negative cycle stop.

iii. Using the distances as a potential modify all of the edges in the graph and run Dijkstras from every vertex.
3. **Coloring Dynamically.** Given a graph $G = (V, E)$, a (proper) $k$-coloring is a coloring of the vertices of the graph using $k$ colors, so that the endpoints of each edge get distinct colors. An equivalent definition is that the vertices having any single color form an independent set.

This problem is NP-hard for $k \geq 3$, so we explore fast exponential-time algorithms.

(a) Show that the problem of 2-coloring a graph can be solved in linear time.

**Solution:** A graph is 2-colorable if and only if it is bipartite. That can be checked using DFS/BFS.

(b) The naive algorithm to $k$-color a graph takes $k^n$ time. Give an algorithm that runs in time $O(k^3 n)$. [A simpler bound is $O(k^4 n)$.]

**Solution:** Let $G[X]$ be the “induced” graph obtained by just keeping the vertices in $X$, and the edges that go between them. Let $T(X, \ell) = 1$ when the graph $G[X]$ can be colored using at most $\ell$ colors, and $T(X, \ell) = 0$ otherwise.

Clearly, $T(\emptyset, 0) = 1$, and $T(S, 0) = 0$ for all non-empty $S$. Moreover,

$$T(X, \ell) = 1 \iff \text{there is some independent set } Y \subseteq X, \text{ such that } T(X \setminus Y, \ell - 1) = 1.$$  

So to calculate $T(X, \ell)$, we can enumerate over all subsets $Y$, and use the values for $T(\cdot, \ell - 1)$. This takes time $2^{|X|}$.

Hence the total time to fill all entries $T(\cdot, \ell)$ is — there are $2^n$ choices of $X$ and each takes at most $2^n$ time, giving $4^n$. One can do better: the total time is, in fact,

$$\sum_{X \subseteq V} 2^{|X|} = \sum_{k=1}^{n} \binom{n}{k} 2^k = (1 + 2)^n.$$  

Summing this for $\ell = 1, 2, \ldots, k$, gives us runtime $O(k3^n)$.

4. **Bin-Packing.** You are given a collection of $n$ items, each item has size $s_i \in [0, 1]$. You have many bins, each of unit size, and you want to pack the $n$ items into as few bins as possible. (Each bin can take a subset of items, whose total size is at most 1.)

Show that you can solve this problem in time $O(n3^n)$. (Hint: subset DP.)

**Solution:** Consider the following subproblems: $P(X, b)$ which represents the question "Can we pack the elements of $X$ into $b$ bins?"

The bases case are $P(\emptyset, 0) = 1, P(X, 0) = 0$ for any non-empty $X$.

We can solve this for some set $X$ by doing the following: For every subset of $X$ that can be fit into a single bin, $Y$, ask $P(X \setminus Y, b - 1)$. Any of these subproblems returning true means we return true.

The runtime is basically the same as the previous problem, but instead of having $k$ colors, we now can have no more than $n$ bins, which gives us a runtime of $O(n3^n)$.