Recap of this week’s lectures:

- Shortest paths: Bellman-Ford, Matrix-product, Floyd-Warshall, Johnson
- Flow: Ford-Fulkerson

Minimum Product Path:
We are given a directed graph where all edges have some positive real weight. Given two vertices a and b, let’s consider paths from a to b. Design an $O(NM)$ algorithm to calculate the minimum possible product of the weights of edges on a path from a to b. Assume there are no cycles that multiply to a value less than 1.

Solution:

We want to minimize $\prod w_i$. This is equivalent to minimizing $\ln(\prod w_i) = \sum \ln w_i$. So we replace every weight $w_i$ in the graph with $\ln(w_i)$, find the $d$, the length of the shortest sum path from a to b. Since there are negative weights in the graph, we should use Bellman Ford. The answer will be $e^d$.

The Traveling Salesperson Problem:
Given a connected graph with $N$ vertices, we want to consider paths (starting and ending anywhere) that visit all vertices. The path may visit nodes and edges more than once. What is the length of the shortest such path? (The problem solved in the lecture notes is a bit different. That version requires finding a cycle that touches all the vertices at least once. Here any path that covers all the vertices is sufficient.)

Solution:

First we compute the all pairs distances between the vertices. Let $d(u, v)$ denote the shortest path distance between vertices $u$ and $v$. Let $DP[S][u]$, where $u \in S$, be the shortest path visiting all nodes in $S$ that ends at $u$. The base cases are: for all nodes $u$, $DP[\{u\}][u] = 0$.

The recurrence is:

$$DP[S][u] = \min_{v \in S} (DP[S \setminus u][v] + d(v, u))$$

where $d(v, u)$ is the length of the shortest path from $v$ to $u$. The runtime is $O(n^22^n)$. In practice, a subset of the nodes is represented as a binary string of length $N$. 
**Bin-Packing:**
You are given a collection of \( n \) items, and 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 \( O(4^n) \). Then improve this bound to \( O(3^n) \).

**Solution:** Define \( DP(X) \) to be the minimum number of bins needed to pack \( X \), an arbitrary non-empty subset of the items. Here is the recurrence for \( DP() \):

\[
DP(X) = \begin{cases} 
1 & \text{if } X \text{ can be packed into one bin} \\
\min_{Y \subseteq X, Y \neq X, Y \neq \emptyset} DP(Y) + DP(X \setminus Y) & \text{otherwise}
\end{cases}
\]

There are \( 2^n \) possible sets \( X \) and \( 2^n \) possible subsets \( Y \) of \( X \). So, an upper bound on the runtime of this algorithm is \( O(4^n) \).

We can get a tighter bound of \( O(3^n) \) on the runtime, because \( \sum_{A \subseteq [1, 2, \ldots, n]} \sum_{B \subseteq A} 1 = 3^n \). You can show this by bijecting iterations of inner loop to base 3 strings of length \( n \). The bijection represents each element in \([1, 2, \ldots, n] \) as a digit in the string. For each element, its digit is 2 if it’s in \( B \), 1 if it’s in \( A \setminus B \), and 0 otherwise.

**Bonus:** Can you solve bin-packing in \( O(n2^n) \) time?

**Solution:**
In brief, for all subsets \( S \) of the items, define \( DP(S) \) as the minimum number of bins to hold \( S \), and \( R(S) \) to be, out of all ways to pack \( S \) into \( DP(S) \) bins, the maximum possible remaining space in a single bin. Now clearly

\[
DP(S) = \min_{i \in S} \left( DP(S \setminus i) + \begin{cases} 
0 & s_i \leq R(S \setminus i) \\
1 & \text{otherwise}
\end{cases} \right)
\]

and you can convince yourself that

\[
R(S) = \max_{i \in S} \left( \begin{cases} 
R(S \setminus i) - s_i & s_i \leq R(S \setminus i) \\
1 - s_i & \text{otherwise}
\end{cases} \right)
\]
Example of running Ford-Fulkerson: Here is a problem from the midterm in 2013. Consider the graph below.

Run Ford-Fulkerson and show the final residual graph. Also what is the maximum flow?

Solution: Max flow is 12.
**Hall’s Marriage Theorem.** Given a bipartite graph $G = (L, R, E)$, we want to find the maximum matching in $G$.

(a) Add a new source $s$ and target $t$. Add unit capacity directed edges from $s$ to vertices in $L$, and from vertices in $R$ to $t$. Direct edges in $E$ towards $R$, make them infinite capacity.

Show a correspondence between integral $s$-$t$ flows in this flow network, and matchings in $G$. (Hence the maximum matching in $G$ equals the maximum flow and minimum $s$-$t$ cut in this network.)

(b) For any set $S \subseteq L$, let $N(S) = \{v \in R \mid \exists u \in S, (u, v) \in E\}$ be the “neighbors” of $S$. Hall’s Marriage theorem says: the size of the maximum matching in $G$ equals $|L|$ if and only if for each subset $S \subseteq L$, $|N(S)| \geq |S|$. Deduce Hall’s theorem from part (a), and the max-flow min-cut theorem.

**Solution:**

Part (a):

Very briefly:

Consider a flow of size $M$. Take the set of edges between $L$ and $R$ that are sending 1 unit of flow. These edges form a valid matching of size $M$.

Now consider a matching of size $M$. Send one unit of flow on each edge in the matching, as well as on all edges connecting $s$ to a matched vertex in $L$ or a matched vertex in $R$ to $t$. This is valid a flow of size $M$.

Part (b):

For the if-and-only-if proof, we need to prove the two directions.

For one direction: suppose the max-matching has size $M = |L|$. Then clearly each subset $S$ has $|N(S)| \geq |S|$, since the neighborhood $N(S)$ must contain all the nodes in $R$ that the vertices in $S$ are matched to.

Now the other direction: suppose the max-matching has size $M < |L|$. By part (a), the $s$-$t$ min-cut in $G$ has capacity $< |L|$. Look at this minimum cut $(A, B)$ with $s \in A, t \in B$; the total capacity of arcs from $A$ to $B$ is $M < |L| < \infty$. So none of the infinite capacity edges can go from $A$ to $B$. In other words, all arcs from $A$ to $B$ are either $s$-to-$L$ edges, or $R$-to-$t$ edges, each of unit capacity. (Infinite capacity arcs could go from $B$ to $A$, of course.)
Define the sets \( S := A \cap L \), and \( T := A \cap R \) as being the intersection of \( A \) with the left and right, respectively. We have that \( M = |L \setminus A| + |A \cap R| = (|L| - |S|) + |T| \). (Why? Each unit of capacity cut by \((A, B)\) corresponds to a node in \( L \setminus A \), or in \( L \cap R \), as in the figure above.)

Now since \( M = (|L| - |S|) + |T| < |L| \), we get that \( |T| < |S| \). Moreover, all arcs leaving \( S \) have infinite capacity, so they must stay within \( A \). This means \( N(S) \subseteq (A \cap R) = T \). So \( |N(S)| \leq |T| < |S| \).

This completes the proof of the second direction, and hence proves Hall’s theorem.
Bonus: aMAZEing Path Finding Skills

You have been magically transported to a maze! The maze is represented as a directed, weighted graph, of \( n \) nodes. You are at node \( s \), and want to find the shortest path to node \( t \).

However, this is no ordinary maze. You have been given a button. Each time you press the button, the walls of the maze change. In other words, you stay at the same node, but the edges between nodes may change in weight, or even be created or deleted. The \( n \)th time you press the button, the exit to the maze will forever be closed off and you’ll be stuck!

You always have full knowledge of the current layout of the maze. However because of your poor memory, you only have \( O(n^2) \) memory in total for your algorithm. This isn’t enough to store all previous layouts of the maze, but it’s enough to create DP tables!

You realize you still have homework left to do and need to escape as fast as possible. Devise an \( O(n^4) \) algorithm to calculate the shortest amount of time required to escape the maze. Note that the max number of edges per map is \( O(n^2) \).

Solution:

Define \( G_d \) as the graph on depth \( d \), where \( G_1 \) is the initial layout of the maze.

Let \( DP[d][u][k] \) be the shortest path to node \( u \) in \( G_d \), using \( k \) edges in \( G_d \).

\[
DP[d][u][k] = \begin{cases} 
0 & d = 1, k = 0 \text{ and } u = s \\
\infty & d = 1, k = 0 \text{ and } u \neq s \\
DP[d - 1][u][n - 1] & d > 1 \text{ and } k = 0 \\
\min(DP[d][u][k - 1], \min_{(u,v,w) \in E(G_d)}(DP[d][v][k - 1] + w)) & k > 0 
\end{cases}
\]

We can optimize the memory to \( O(n^2) \) because once we have calculated \( DP[d] \), we no longer require \( DP[d - 1] \) for future calculations. Thus we only need to store in memory at most two tables of size \( O(n^2) \). In fact, it is possible to apply the same logic to the last parameter to bring the memory to \( O(n) \), but this isn’t required.

Another way to view this algorithm is that for every layout of the maze, we are doing Bellman Ford at each level, except that instead of initializing all \( DP[u][0] = \infty \), we initialize \( DP[u][0] \) with the length of shortest path to \( u \) in the previous layout.