DP and Graph Algorithms
Problem Set 2 – CS 6515/4540 (Fall 2025)

This problem set is due on **Tuesday October 1st**. Submission is via Gradescope. Your solution must be a typed pdf (e.g. via LaTeX) – no handwritten solutions.

4 DP: Balancing Array

Given an array of n **nonnegative** integers A and an integer k, we want to partition A into k contiguous nonempty subarrays such that the subarray with the largest sum is minimized. Let the minimum achievable largest sum be S. Provide an algorithm to compute S as efficiently as possible using Dynamic Programming.

Example: $A = [1, 4, 6, 2, 5], k = 3$. In this case, the most balanced subarray partition of A is $[1, 4], [6], [2, 5]$ and we have $S = \max(1 + 4, 6, 2 + 5) = 7$.

1. Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...
2. State recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.
3. Analyze the running time of your algorithm. It should be $O(kn \log n)$

Remark: There is a more efficient way to solve this without using DP that eliminates any dependency on k.

Solution Sketch: Basically $dp[i][j]$ stores the answer to the Balancing Array problem for first i elements of $A = A[0, i]$ and $k = j$. The recurrence is just

$$dp[i][j] = \min_{i'<i} \max\left(\text{sum}(A[i' + 1, i]), dp[i'][j - 1]\right)$$

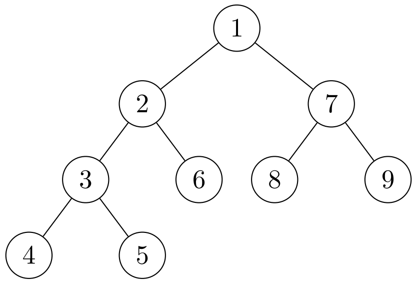
We want to efficiently find the index i' that minimizes the max function above. Note that we can just binary search for the exact index i' where $dp[i'][j-1]$ is no longer greater than sum$A[i' + 1, i]$ (i.e. the j-th partition begins to dominate). The minimum quantity will be at i' or $i' + 1$. Hence we can get $O(kn \log n)$ time by evaluating the recurrence in $O(\log n)$ time via binary search. Binary search works since $\text{sum}(A[i' + 1, i])$ decreases as i' increases but $dp[i'][j - 1]$ increases as i' increases.

5 DP: Minimum Lights

The Georgia Tech campus spans over n academic buildings, with pathways that interconnect resembling a tree $T = (V, E)$. The campus police want to ensure all pathways are illuminated at night. Lights can be installed on any building $B \in V$, and when done so, all pathways/edges connected to B are lighted.

Your goal is to develop a strategy using Dynamic Programming to find the minimum number of lights needed to ensure the entire campus is well-lit at night. For this, you are given an unweighted undirected tree structure with n vertices and m edges, where each node represents an academic building and the edges represent pathways. Find some $S \subseteq V$ such that every edge $e \in E$ is incident to at least one vertex in S, and $|S|$ is minimized. The running time of your algorithm should be $\mathcal{O}(|V|)$.

Example Input:



Example Output: 3. The buildings that need to lit up are {2, 3, 7}.

1. Define the entries of your table in words. E.g. $T(i)$ or $T(i, j)$ is ...

   The DP table $T(v_i)$ will keep track of the minimum number of lights needed to light the subtree rooted at v_i. Specifically, $T(v_i)$ will be a tuple/2-D array keeping track of ($T(v_i)(1) = x$, $T(v_i)(0) = y$) where x and y are the minimum lights required to ensure the subtree is lit if we include v_i or exclude v_i, respectively.

2. State recurrence for entries of the table in terms of smaller subproblems. Briefly explain in words why it is correct.

   The base case will be for every leaf node, $T(v) = (1, 0)$ because $Include_v = 1$, $Exclude_v = 0$ as there are no children for the leaf node to consider for the exclude case.

   $T(v_i)(1) = 1 + \sum_u \min(T(u)(0), T(u)(1))$ and
   $T(v_i)(0) = \sum_u T(u)(1)$,
   where in both cases u is a child of v_i and we are summing over all the child nodes of v_i.

   The given recurrence works.

   (a) Including v_i: This means we are placing a light on building v_i. The immediate consequence is that all edges/pathways connected to v_i are lighted. For its children, since the pathway to them is already lighted, they can either have a light or not. Thus, we consider the minimum of the two cases for each child. The logic being, we are trying to find the optimal solution, so if a child doesn't need to be lit (because it's cheaper overall to light another nearby node), we'll take that path. The sum of all these minimums gives us the total number of lights needed if we decide to place a light at v_i.

   (b) Excluding v_i: This means we are not placing a light on building v_i. In this scenario, for the campus to be well-lit, all children of v_i must have lights since the pathway from v_i to each child must be illuminated. Hence, we sum up the number of lights for all children when they are included.

3. Analyze the running time of your algorithm.

   The algorithm essentially computes the DP values for every node in the tree twice (once for when the node is included and excluded). As there are $|V|$ nodes, and each computation involves examining all the children of a node. Note that each node is a child of at most one node, so the overall runtime is $\mathcal{O}(|V| + |E|)$. Since $|E| = |V| - 1$ for a tree, the runtime is effectively $\mathcal{O}(|V|)$.

6 Knapsack and MDP

6.1 Knapsack

Recall the Knapsack problem where we have budget B and n items labeled i where $1 \le i \le n$, and item i has weight $s_i \le B$ and valuation v_i, and we want to compute the subset with the maximum total valuation $\sum_{i \in I} v_i$ of all subsets $I \subseteq [n]$ such that $\sum_{i \in I} s_i \le B$

1. Show that the PTAS given in lecture where we apply the $O(n^2 \max v_i)$ DP algorithm but with valuations truncated to their nearest smaller multiple of K, will give a feasible subset with total valuation V, which satisfies $1 - \frac{nK}{\max v_i} \le \frac{V}{V'} \le 1$, where V' is the optimal valuation of the original problem.
2. Choose the appropriate K in terms of ϵ and $\max v_i$ such that we get $\frac{V}{V'} \ge 1 - \epsilon$ in poly$(n, \frac{1}{\epsilon})$ time.
3. Suppose there exists an approximation algorithm that could, in polynomial (NOT pseudo polynomial) time, get a $1 - \epsilon$ approximation to Knapsack for $\epsilon < \frac{1}{n \max v_i}$. Then, show that $P = NP$.

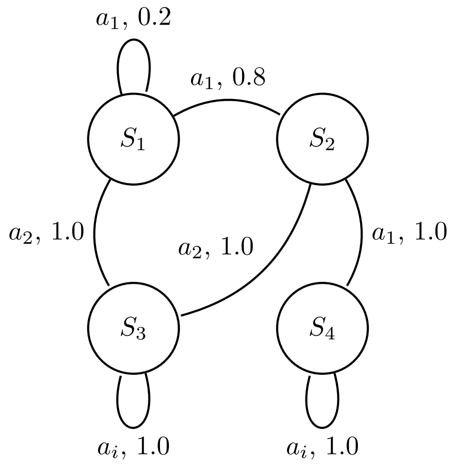
Parts 1 and 2 are a rework of the end of lecture 5 and beginning of lecture 6. The proof details can be found in these lectures.

For part (3), note that since $V' \le n \max v_i$, we have $V' - \epsilon V' = V' - \frac{V'}{n \max v_i} > V' - 1$, and so by $1 - \epsilon$ approximation, we have $V' - 1 < V \le V'$, so we can exactly deduce the value of V' since its integral, solving the Knapsack problem.

6.2 MDP

Consider the MDP with 4 states $\{S_1, S_2, S_3, S_4\}$ and 2 actions $\{a_1, a_2\}$ as given in the figure. Here, the tuple a, p on an edge means that if we perform action $a \in \{a_1, a_2\}$ at the start state then we end at the destination state with probability p. (When we write a_i in the figure that means this is true for both a_1 and a_2.)13 Moreover, we receive reward 10 on pulling any action from S_3, we receive reward 100 on pulling any action from S_4, and we receive reward 0 on pulling any action from S_1 or S_2.

Suppose we start at S_1, then what is the optimal policy for time horizon $T = 2$ (i.e., when you are only allowed 2 pulls)? Justify your answer in a few words.



Since we are forced to pull from S_1 on the first time step and S_2 yields zero reward for pulling an action, we should go to S_3 with 100% probability using action a_2 at S_1, $T = 1$. Then at $T = 2$, we can choose any action from S_3 and pull the reward of 10.

7 Walking on the Plane

Consider n points on the Euclidean plane. Each point is the center of a dynamically changing circle whose radius grows linearly with time, i.e. the radius of each circle at time t is t. Given two vertices u and v, give an algorithm that finds the earliest time t at which u and v become connected - i.e. you can freeze time and walk from u to v on the Euclidean plane and be inside at least one circle at any point in your path. Provide the time complexity and justify the correctness of your algorithm. It must run in $\widetilde{O}(n^2)$ time [1]

Remark: This problem can be solved in $O(n \log n)$ time by computing the Delaunay Triangulation.

Solution Sketch: We can guaranteed walk from a vertex a to vertex b at time $d(a,b)/2$. Thus, consider the complete graph on n vertices where the weight of edge (a,b) is $d(a,b)/2$. Then, run Dijkstra from u to v to find the smallest path from u to v in terms of maximum weight of an edge in the path - we are guaranteed to be able to traverse all edges in that path at that time since it is the largest possible time.

The important thing to note is that Dijkstra works for any cost function that is non-decreasing; set maximum is an example. Our algorithm runs in $O(n^2 \log n)$ time due to guarantees of Dijkstra.

8 Bipartite Matching: Forbidden Paths

Assume we are given a directed graph $G = (V, E)$ with $V = \{v_1, ..., v_n\}$ and a forbidden set $F = \{f_1, f_2, \ldots, f_k\} \subset V$. Define a bipartite graph $\widehat{G} = (L \cup R, \widehat{E})$ as follows: $L = \{v_1, ..., v_n\}$, $R = \{v'_1, ..., v'_n\}$ (so we have two copies of V, one on the left and one on the right side of the bipartite graph). Edge $\{v_i, v'_i\} \in \widehat{E}$ for all $i = 1, ..., n$, and $\{v_i, v'_j\} \in \widehat{E}$ for all $(v_i, v_j) \in E$.

Given two vertices $v_s, v_t \in V$, delete v_s from L and delete v'_t from R. We also delete f_i from L and R for each $f_i \in F$ (when we delete these two vertices, we must also delete any edges from $\widehat{E}$ that v_s and v'_t were connected to).

Problem: Show that a perfect matching exists in $\widehat{G}$ if and only if there exists a directed path from v_s to v_t in G that avoids all vertices in F.

[1] $\widetilde{O}$ means we don't care about sub-polynomial factors like $\log n$. For example, $n^2 \log^2(n) = \widetilde{O}(n^2)$ but $n^{2.3}$ is not.

path avoiding forbidden vertices $\Rightarrow$ perfect matching: Say there exists a path $v_s \to v_{i_1} \to v_{i_2} \to \cdots \to v_{i_m} \to v_t$ that avoids all vertices in the forbidden set $F = \{f_1, f_2, \ldots, f_k\}$. We can construct a perfect matching as follows: Match v_s with v'_{i_1} (cross edge), v_{i_1} with v'_{i_2} (cross edge), and so on, finally matching v_{i_m} with v'_t (cross edge). This way, all vertices in the path and their corresponding copies are matched. For vertices not part of the path and not in the forbidden set F, match each v_ℓ with v'_ℓ (identity edges). This gives us a perfect matching of size $n - k - 1$, since we have $n - k - 1$ vertices on each side of the modified bipartite graph.

perfect matching $\Rightarrow$ path avoiding forbidden vertices: Start with a matching M where we match every available v_ℓ with v'_ℓ (each vertex with its copy using identity edges). The available vertices are those in $V \setminus (\{v_s, v_t\} \cup F)$. This M is not a perfect matching as v_s and v'_t are unmatched. These two are the only unmatched vertices. If there exists a perfect matching, then we can find an augmenting path to increase the size of M. Specifically, we have an augmenting path from v_s to v'_t. This augmenting path cannot pass through any forbidden vertices since all vertices in F were deleted from both sides of the bipartite graph. The augmenting path alternates between cross edges $\{v_i, v'_j\}$ (where $i \neq j$) and identity edges $\{v_\ell, v'_\ell\}$. The cross edges in this augmenting path correspond to directed edges in G, giving us a path from v_s to v_t that avoids all forbidden vertices.

Alternate solution for backwards direction: Students can iteratively match and construct the path avoiding forbidden vertices. Start by examining v_s: since v'_s was deleted, v_s must be matched to some other vertex's copy.