CSE 102 Homework 4

Ex.1 (20 points) (Dynamic Programming on Trees)
Let $G = (V, E)$ be an undirected graph. If G is connected and acyclic, then it is called a tree. A subset I of the vertices V is called an **independent set** if no two vertices of I are adjacent. Assume that a positive weight $w(i)$ is associated with each vertex i. We define the weight $w(I)$ of an independent set I to be the sum of the weights of all the vertices in I. That is, $w(I) = \sum_{i \in I} w(i)$. Further, an independent set is called a maximum weight independent set if it has the maximum possible weight among all independent sets. Give an $O(n)$ time algorithm, which finds the maximum weight independent set of a graph G for the case that G is a tree. As always, justify why it is correct and its runtime. (Hint: we solved the same problem in class when we were doing dynamic programming, for the case that the graph was a path, instead of a tree.)

Solution.

Detailed Dynamic Programming Algorithm for MWIS in Trees

DP Formulation

Let $T = (V, E)$ be our tree with n vertices, and $w(v)$ be the weight of vertex v. Define $DP[v][0]$ and $DP[v][1]$ for each vertex $v \in V$:

- $DP[v][0]$: Maximum weight of independent set in subtree rooted at v, excluding v
- $DP[v][1]$: Maximum weight of independent set in subtree rooted at v, including v

Recursive Relations

For a non-leaf vertex v with children $c_1, c_2, \ldots, c_k$:

$$DP[v][0] = \sum_{i=1}^{k} \max(DP[c_i][0], DP[c_i][1])$$

$$DP[v][1] = w(v) + \sum_{i=1}^{k} DP[c_i][0]$$

Base Cases

For a leaf vertex v:

$$DP[v][0] = 0$$

$$DP[v][1] = w(v)$$

Algorithm

Dynamic Programming for MWIS in Trees

1. Choose an arbitrary root r for T.
2. Initialize a 2D array DP to 0.
3. Initialize an array *choice* to 0 (for reconstruction).
4. Call the function **ComputeDP**(r, T, w, DP, *choice*).
5. Compute the maximum weight: $maxWeight = \max(DP[r][0], DP[r][1])$.
6. Call the function **ReconstructIS**(r, DP, *choice*) to get the independent set IS.
7. Return $maxWeight$ and IS.

ComputeDP Function

1. If v is a leaf:
   - Set $DP[v][0] = 0$
   - Set $DP[v][1] = w(v)$
2. Else:
   - Set $DP[v][1] = w(v)$
   - For each child c of v:
     (a) Call **ComputeDP**(c, T, w, DP, *choice*)
     (b) Update $DP[v][0] = DP[v][0] + \max(DP[c][0], DP[c][1])$
     (c) Update $DP[v][1] = DP[v][1] + DP[c][0]$
     (d) If $DP[c][1] > DP[c][0]$, set $choice[c] = 1$

Reconstruction Algorithm

To reconstruct the actual independent set, we need to keep track of our choices during the DP computation. We use an additional array *choice* where $choice[v] = 1$ if we chose to include vertex v in the optimal solution for its parent, and 0 otherwise.

1. Initialize IS as an empty set.
2. Call the function **ReconstructISHelper**(v, DP, *choice*, IS, true).
3. Return IS.

ReconstructISHelper Function

1. If $canInclude$ and $DP[v][1] \geq DP[v][0]$:
   - Add v to IS.
   - For each child c of v, call **ReconstructISHelper**(c, DP, *choice*, IS, false).
2. Else:
   - For each child c of v, call **ReconstructISHelper**(c, DP, *choice*, IS, true).

Explanation of Reconstruction

We start at the root with the option to include it ($canInclude = true$). At each vertex v:

- If we can include v and it's optimal to do so ($DP[v][1] \geq DP[v][0]$), we add v to the independent set and recursively process its children with $canInclude = false$.
- Otherwise, we don't include v and recursively process its children with $canInclude = true$.

This process ensures we reconstruct the optimal independent set that corresponds to our DP solution.

Time and Space Complexity

- **Time Complexity:** $O(n)$
  - The algorithm visits each node once during the DFS traversal.
  - At each node, we perform a constant amount of work (O(1)) for initialization.

- The work done for processing children at each node adds up to O(n) across all nodes, because:
  - Each edge in the tree is considered exactly once (when processing the parent node).
  - The number of edges in a tree is always n-1.
- The total work is therefore linear in the number of nodes.

In general graphs, we might need to check all edges for each vertex to avoid revisiting nodes. In trees, there are no cycles, so we don't need this extra checking.

- **Space Complexity:** $O(n)$
  - We store two values for each node in the DP array.

Example

Consider this small tree:

1(4)
/|
2(7)3(2)4(6)
/
5(1)6(3)7(5)

Numbers in parentheses are weights. Let's solve bottom-up:

- Node 5: $DP[5][0] = 0$, $DP[5][1] = 1$
- Node 6: $DP[6][0] = 0$, $DP[6][1] = 3$
- Node 7: $DP[7][0] = 0$, $DP[7][1] = 5$
- Node 2: $DP[2][0] = 0$, $DP[2][1] = 7$
- Node 3: $DP[3][0] = \max(DP[5][1], DP[6][1]) = 3$, $DP[3][1] = 2 + DP[5][0] + DP[6][0] = 2$
- Node 4: $DP[4][0] = DP[7][1] = 5$, $DP[4][1] = 6 + DP[7][0] = 6$
- Node 1: $DP[1][0] = \max(DP[2][1], DP[2][0]) + \max(DP[3][1], DP[3][0]) + \max(DP[4][1], DP[4][0]) = 7 + 3 + 6 = 16$,
  $DP[1][1] = 4 + DP[2][0] + DP[3][0] + DP[4][0] = 4 + 0 + 3 + 5 = 12$

Thus, the maximum weight of an independent set in this tree is $\max(DP[1][0], DP[1][1]) = \max(16, 12) = 16$.

Ex.2 (20 points) (DNA comparisons)
In this problem, we explore some common DNA problems that arise in Biology. Given two strings of characters X (of length n) and Y (of length m), we care to compute the **length** of their longest common subsequence. If there is no common subsequence, the answer is of course 0.

A **subsequence** of a string is a new string generated from the original string with some characters deleted without changing the relative order of the remaining characters. A **common subsequence** of two strings is a subsequence that is common to both strings.

Examples: "ace" is a subsequence of "abcde". Another example is that "abc", "abg", "bdf", "aeg", "acefg", all are subsequences of "abcdefg".

- Describe an algorithm that finds the length of the longest common subsequence between two strings X and Y. What's the runtime of your algorithm? Your algorithm should not be of exponential time and you should clearly define any subproblems needed and/or any recursions that you write down (Hint: Dynamic Programming).

- Give a reconstruction algorithm that finds and outputs the longest common subsequence between X and Y.

- Given two strings X (of length n) and Y (of length m), we ask you to find the length of the **shortest string** that has both X and Y as subsequences. Here are two examples: If X is "beek", and Y is "eke", then the answer is 5, because the string "beeke" has both string "beek" and "eke" as subsequences and is shortest possible. For another example, take X to be "AGGTAB", and Y to be "GXTXAYB", then the answer is 9, because the string "AGXGTXAYB" has both strings "AGGTAB" and "GXTXAYB" as subsequences and is shortest possible. (Hint: try to figure out what's the connection with the longest common subsequence problem.)

  (**Bonus:** what would change if we asked to find the length of the **shortest string** that has both X and Y as **substrings** instead?).

Solution.

Exercise 2: DNA Comparisons

Problem Statement

Given two strings X (of length n) and Y (of length m), we need to solve several related problems involving subsequences and supersequences.

Definitions

- A **subsequence** of a string is a new string generated from the original string by deleting some characters without changing the relative order of the remaining characters.
- A **common subsequence** of two strings is a subsequence that is common to both strings.
- The **Longest Common Subsequence (LCS)** is the longest subsequence common to both strings.

Part 1: Length of Longest Common Subsequence

Algorithm Description

We use dynamic programming to find the length of the LCS.

Recurrence Relation

Let $dp[i][j]$ be the length of the LCS of prefixes $X[1...i]$ and $Y[1...j]$.

$$dp[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ dp[i-1][j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(dp[i-1][j], dp[i][j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

Pseudocode

1. Initialize $dp[0...n][0...m]$ with 0.
2. For $i \leftarrow 1$ to n:
   (a) For $j \leftarrow 1$ to m:
      i. If $X[i] = Y[j]$:
         A. $dp[i][j] \leftarrow dp[i-1][j-1] + 1$
      ii. Else:
         A. $dp[i][j] \leftarrow \max(dp[i-1][j], dp[i][j-1])$
3. Return $dp[n][m]$

Intuition

The key insight is that for each pair of characters (one from X, one from Y), we have two choices:

- If they're the same, we include this character in our LCS and move to the next pair.
- If they're different, we take the best of two options: skip the character from X or skip the character from Y.

Runtime Analysis

- **Time Complexity:** $O(nm)$
  - We fill a 2D table of size $n \times m$.
  - For each cell, we perform constant-time operations.
  - Total operations: $n \times m \times O(1) = O(nm)$
- **Space Complexity:** $O(nm)$
  - We store a 2D table of size $n \times m$.

Optimization Note

We can optimize space to $O(\min(n, m))$ by only storing two rows at a time, as each cell only depends on the previous row and the current row.

Part 2: Reconstruction of LCS

Algorithm Description

We reconstruct the LCS by tracing back through the *dp* table.

Pseudocode

1. Initialize $i \leftarrow n, j \leftarrow m$.
2. Initialize $lcs \leftarrow$ empty string.
3. While $i > 0$ and $j > 0$:
   (a) If $X[i] = Y[j]$:
      i. Prepend $X[i]$ to lcs.
      ii. $i \leftarrow i - 1, j \leftarrow j - 1$.

(b) Else if $dp[i-1][j] > dp[i][j-1]$:

i. $i \leftarrow i - 1$.

(c) Else:

i. $j \leftarrow j - 1$.

4. Return lcs.

Intuition

We trace back through our DP table, making decisions based on how we arrived at each cell:

- If characters match, we include them in our LCS.
- If not, we move in the direction of the larger value (up or left).

Runtime Analysis

- **Time Complexity:** $O(n + m)$
  - Max steps $= n + m$
- **Space Complexity:** $O(n + m)$
  - LCS length $\leq \min(n, m)$

Part 3: Shortest Supersequence

Problem

Find the length of the shortest string that has both X and Y as subsequences.

Solution

The length of the shortest supersequence is:

$$\text{length\_of\_shortest\_supersequence} = n + m - \text{LCS\_Length}(X, Y)$$

Intuition

The shortest supersequence includes all characters from both strings, but characters in the LCS should appear only once.

Runtime Analysis

- **Time Complexity:** $O(nm)$
- **Space Complexity:** $O(nm)$ (can be optimized to $O(\min(n, m))$)

Bonus: Shortest String with X and Y as Substrings

When finding the length of the shortest string that has both X and Y as substrings, we initially might try to adapt our previous dynamic programming approach:

Initial Attempt

Let $dp[i][j]$ be the length of the shortest string that contains $X[1...i]$ and $Y[1...j]$ as substrings. Recurrence relation:

$$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 \\ dp[i][j-1] + 1 \\ dp[i-k][j-k] + \max(i, j) - k \quad \text{if } X[i-k+1 \dots i] = Y[j-k+1 \dots j] \end{cases}$$

However, this approach fails because:

- It doesn't consider all possible overlaps between X and Y.
- The optimal solution might involve interleaving substrings in complex ways.

Complexity Analysis

Exploring all possible overlaps and interleaving leads to:

- $O(2^n)$ possible combinations to check, where $n = \max(|X|, |Y|)$.
- Each combination requires $O(n)$ time to verify and construct.

Thus, a naive exact algorithm would have $O(n \cdot 2^n)$ time complexity.

Conclusion

This analysis reveals that:

- The problem is the Shortest Common Superstring (SCS) problem.
- It's NP-hard, explaining our failure to find a polynomial-time solution.
- Exact solutions require exponential time.

Alternative Approaches

Given the NP-hardness, we consider approximation algorithms:

1. **Greedy Overlap Method:**
   - Repeatedly merge strings with maximum overlap.
   - Provides a 2.5-approximation of the optimal solution.
   - Time Complexity: $O(n^2)$ for two strings.
2. **Suffix-Prefix Overlap:**
   - Find the maximum suffix-prefix overlap between X and Y.
   - The length of the shortest superstring is at least $n + m - overlap$.
   - Time Complexity: $O(n + m)$ using suffix arrays.

Ex.3 (10 points) (Smallest Product Paths) In class we saw the shortest path problem. Here we examine a variant where we are given a weighted undirected graph G (assume weights are positive integers) and a starting vertex s, and our goal is to compute the **cheapest** paths from s to all other vertices. However, the cost of a path is **not** the sum of its edge weights, but rather the **product** of its edge weights.

Give a simple algorithm to solve this problem with a runtime similar to Dijkstra's algorithm, and justify your algorithm. (A correct answer shouldn't be more than 3-4 lines.)

Solution.

Exercise 3: Smallest Product Paths

Problem Statement

Given a weighted undirected graph G with positive integer weights and a starting vertex s, compute the cheapest paths from s to all other vertices, where the cost of a path is the product of its edge weights.

Algorithm

We can modify Dijkstra's algorithm to solve this problem efficiently:

1. Transform the graph G by taking the natural logarithm of each edge weight w: $w' = \log(w)$.

2. Apply Dijkstra's algorithm on the transformed graph, starting from s.
3. Initialize distances: $d[s] = 0$, $d[v] = \infty$ for $v \neq s$.
4. Use a priority queue Q for vertices, keyed by their d values.
5. For each edge (u, v), update $d[v] = \min(d[v], d[u] + w'(u, v))$.
6. After Dijkstra's algorithm completes, exponentiate the results: $d[v] = e^{d[v]}$.

Justification

This approach works because:

- $\log(a * b) = \log(a) + \log(b)$, so products become sums in the log form.
- The logarithm is monotonic, preserving the order of path costs.
- Exponentiating the results converts back to the original form.

Time Complexity

The time complexity is the same as Dijkstra's algorithm:

$$O((V + E) \log V) \approx O(V \log V) \approx O(n \log n)$$

Where V is the number of vertices and E is the number of edges.