Data Structures and Algorithms Final Report 44251017 Huang Jiahui

Q1

(1) Describe an efficient algorithm that finds a maximum weight tour *r*. If you use an algorithm design paradigm, show its name.

The problem can be solved efficiently using Dynamic Programming on Trees. The core idea is to perform a single traversal of the tree (specifically, a post-order traversal or Depth First Search). For each node, we compute the longest path starting at that node and going downwards, while simultaneously tracking the longest path found anywhere in the tree so far.

Algorithm Design Paradigm: Dynamic Programming on a Tree.

```
// Assume a TreeNode structure exists with:
//   int weight;
//   List<TreeNode> children;

// --- Global Variable ---
// Used to track the maximum weight found throughout the recursion.
int max_weight_so_far;


// --- Main Function ---
// Description: The entry point to start the algorithm.
FUNCTION findMaximumWeightTour(TreeNode root):
    // Initialize the global maximum to a very small value.
    max_weight_so_far = Integer.MIN_VALUE;

    // Call the recursive helper to traverse the entire tree.
    dfsHelper(root);

    // Return the final recorded maximum weight.
    RETURN max_weight_so_far;


// --- Recursive Helper Function (DFS) ---
// Description: Processes each node in a post-order fashion to calculate paths and update the global max.
// Returns: The weight of the longest "single-branch" path starting from 'node' and going downwards.
FUNCTION dfsHelper(TreeNode node):
    // Base case: If the node doesn't exist, its contribution to any path is 0.
    IF (node is null):
        RETURN 0;

    // Recursively call for all children.
```

```
    List<Integer> child_down_paths = new ArrayList<>();
    FOR EACH TreeNode child in node.children:
        // Calculate the downward path weight from the child.
        int path_from_child = dfsHelper(child);
        // Add it to our list, but don't consider negative paths (treat them as 0).
        child_down_paths.add( max(0, path_from_child) );

    // Find the two longest downward paths coming from the children.
    Sort child_down_paths in descending order;

    int max1 = 0;
    int max2 = 0;
    IF (child_down_paths is not empty):
        max1 = child_down_paths.get(0);
    IF (child_down_paths has more than 1 element):
        max2 = child_down_paths.get(1);

    // Calculate the path "through" the current node and try to update the global max.
    int path_through_node = node.weight + max1 + max2;
    max_weight_so_far = max(max_weight_so_far, path_through_node);

    // Return the longest "single-branch" downward path from this node for its parent to use.
    RETURN node.weight + max1;
```

(2) Demonstrate how your algorithm finds maximum weight tour on the above tree.

We use the correct tree structure and perform a post-order traversal. Let `M` be `max_weight_so_far`, initialized to `Integer.MIN_VALUE`.

1. Leaves:
    - `dfsHelper(40)` returns 40. M = 40.
    - `dfsHelper(20)` returns 20. M = 40.
    - `dfsHelper(5)` returns 5. M = 40.
    - `dfsHelper(32)` returns 32. M = 40.
    - `dfsHelper(22)` returns 22. M = 40.
    - `dfsHelper(60)` returns 60. M = 60.
    - `dfsHelper(3)` (leaf) returns 3. M = 60.
2. Internal Nodes (in post-order):
    - Node 2: Child is 40. Returns `w(2) + 40 = 42`. M = max(M, 42) = 60.
    - Node 10 (left): Children are 2, 20. `dfsHelper` returned 42 and 20.

        - `max1`=42, `max2`=20.
        - `path_through_10(L)` = w(10) + 42 + 20 = 72. M = max(60, 72) = 72.
        - Returns `w(10) + 42 = 52`.
    - Node 10 (right): Child is 32. Returns `w(10) + 32 = 42`. M = max(M, 42) = 72.
    - Node 6: Children are 22, 60, 3(leaf). `dfsHelper` returned 22, 60, 3.
        - `max1`=60, `max2`=22.
        - `path_through_6` = w(6) + 60 + 22 = 88. M = max(72, 88) = 88.
        - Returns `w(6) + 60 = 66`.
    - Node 4: Children are 10(right), 6. `dfsHelper` returned 42, 66.
        - `max1`=66, `max2`=42.
        - `path_through_4` = w(4) + 66 + 42 = 112. M = max(88, 112) = 112.
        - Returns `w(4) + 66 = 70`.
    - Node 3 (root): Children are 10(left), 5, 4. `dfsHelper` returned 52, 5, 70.
        - `max1`=70 (from node 4), `max2`=52 (from node 10-left).
        - `path_through_3` = w(3) + 70 + 52 = 125. M = max(112, 125) = 125.
        - Returns `w(3) + 70 = 73`.

The final value for `max_weight_so_far` is 125. This maximum was found when considering the root node `3` as the apex. The path is formed by combining the heaviest downward path from node `4` (`4 -> 6 -> 60`), the heaviest downward path from the left node `10` (`10 -> 2 -> 40`), and the apex node `3` itself.

The path is: 40 - 2 - 10(left) - 3 - 4 - 6 - 60. Weight = 40+2+10+3+4+6+60 = 125.

(3) Prove that your algorithm is correctly finding a tour having maximum weight.

Proof by Contradiction and Induction:

Let `P` be a maximum weight tour (path) in the tree `T`. Any path `P` has a unique "highest" node, which we call the *apex* of the path. The apex is the node on the path that is closest to the root of the tree.

Our algorithm iterates through every node `u` in the tree and considers it as a potential apex. For each `u`, it calculates the weight of the best possible path with `u` as its apex. The maximum of these values is the result.

Let `u*` be the apex of the true maximum weight path `P`. The path `P` consists of `u*` and at most two disjoint downward paths starting from children of `u*`. Let these downward paths be `P1` (starting at child `v1`) and `P2` (starting at child `v2`). The total weight is `w(P) = w(u*) + w(P1) + w(P2)`.

The core of our algorithm is the function `dfsHelper(v)`, which we claim correctly computes the maximum weight of a path starting at `v` and descending into its subtree. We can prove this by induction on the height of the subtrees.

- Base case (height 0): For a leaf `v`, `dfsHelper(v)` returns `w(v)`, which is correct.
- Inductive step: Assume `dfsHelper(c)` is correct for all children `c` of `v`. Then the maximum downward path from `v` must consist of `v` itself plus the maximum downward path from one of its children `c'`. Our algorithm computes this as `w(v) + max(0, max_c(dfsHelper(c)))`. This is correct.

Since `dfsHelper` is correct, when our algorithm reaches the node `u*` (the apex of the true max-weight path `P`), it will have correctly computed `dfsHelper(v1)` and `dfsHelper(v2)`. These must be equal to `w(P1)` and `w(P2)`. If they weren't, it would contradict that `P` is the overall maximum weight path.

Our algorithm calculates the candidate path weight at `u*` as `w(u*) + max1 + max2`, where `max1` and `max2` are the weights of the two heaviest downward paths from its children. This calculation will precisely equal the weight of `P`.

Since the algorithm performs this calculation for *every* node as a potential apex, it is guaranteed to perform it for the true apex `u*`. Therefore, the final value of `max_weight_so_far` must be the weight of the maximum weight path `P`.

(4) Suppose that tree *T* has *n* nodes. Show the worst-case running time of your algorithm.

The running time of the algorithm is determined by the traversal and the work done at each node.

1. **Traversal:** The algorithm is based on a single Depth First Search (or post-order traversal) of the tree. In a tree with `n` nodes and `n-1` edges, a DFS visits each node and each edge a constant number of times. This traversal has a complexity of $O(n + (n-1)) = O(n)$.
2. **Work per Node:** For each node `u`, the algorithm performs the following operations:
   - It receives the return values from all its children.
   - It iterates through these values to find the top two largest. If node `u` has `deg(u)-1` children (where `deg(u)` is its degree), sorting or finding the top two takes time proportional to the number of children, which is $O(deg(u))$.
   - It performs a constant number of additions and comparisons.
3. **Total Time:** The total running time is the sum of the work done at each node during the traversal.

   $T(n) = \sum O(deg(u)) \quad , u \in V$

   In graph theory, the sum of the degrees of all vertices in a graph is equal to twice the number of edges:

   $\sum deg(u) = 2 \mid E \mid$

   For a tree with `n` nodes, the number of edges `|E|` is `n-1`.

   $\sum deg(u) = 2(n-1)$

   Therefore, the total time complexity is $O(2(n-1)) = O(n)$.

> The worst-case running time of the algorithm is O(n), which is linear in the number of nodes.

Q2

Given recurrence:

$$T(n) = \begin{cases} c & \text{n<2} \\ T(n/2) + logn & \text{n}\geq 2 \end{cases}$$

Expansion:

$T(n) = T(n/2) + logn$

$= T(n/4) + log(n/2) + logn$

$= T(n/8) + \log(n/4) + \log(n/2) + \log n$

Continue expanding until $n/2^k < 2$, so $k = log_2 n$.

Sum up all terms:

$T(n) = c + logn + log(n/2) + log(n/4) + \cdots + log2$

$= c + logn + (logn - 1) + (logn - 2) + \cdots + 1$

This is an arithmetic series, and the sum is:

$T(n) = \Theta((logn)2)$

Q3

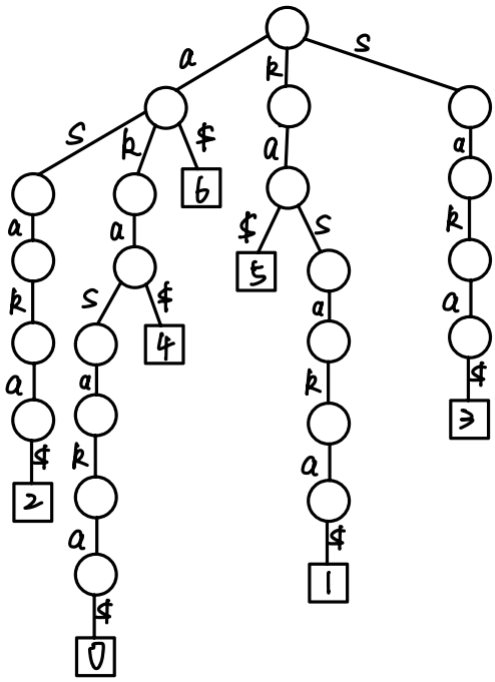
(1) Show the suffix trie for the following string: akasaka

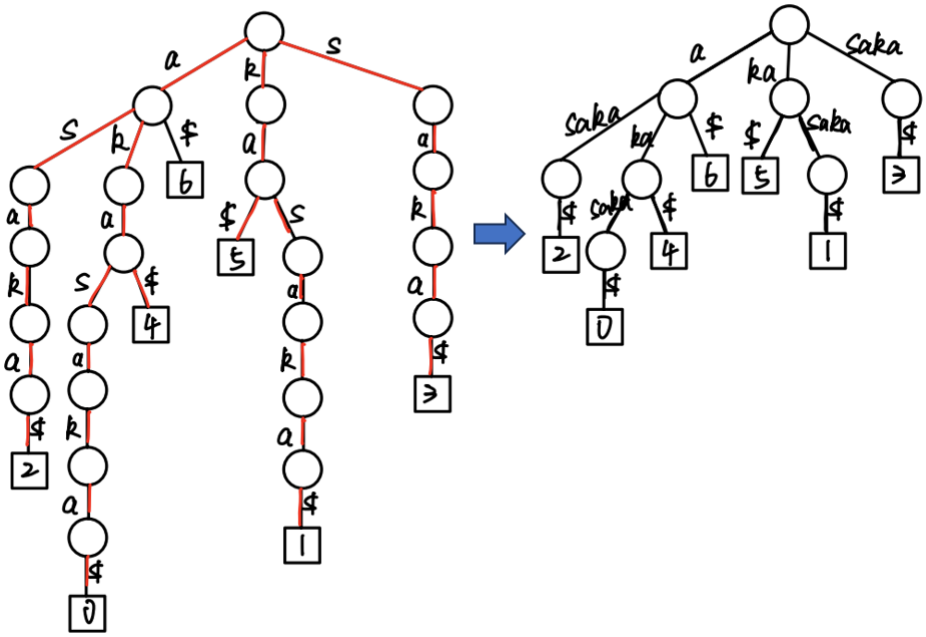
To build a suffix trie, we first list all suffixes of the string. To ensure that each suffix has a unique path to a leaf, we append a special terminal character, `$`, which is not in the alphabet. The string becomes `akasaka$`.

The suffixes are:

1. `akasaka$` (index 0)
2. `kasaka$` (index 1)
3. `asaka$` (index 2)
4. `saka$` (index 3)
5. `aka$` (index 4)
6. `ka$` (index 5)
7. `a$` (index 6)

By inserting these suffixes into a trie, we get the following structure. The numbers in the leaf nodes represent the starting index of the corresponding suffix.





(2) Let aXa be a pattern match query that finds all the substrings 1) starting with “a”, 2) followed by zero or more characters of any kind, and 3) ending with “a”. Show the results of applying query aXa on “akasaka.”

We need to find all substrings of "akasaka" that start with 'a' and end with 'a'. We can find these by manual inspection. The results are listed below, along with their start and end positions (0-indexed) for clarity.

- `aka` (positions 0 to 2)
- `akasa` (positions 0 to 4)
- `akasaka` (positions 0 to 6)
- `asa` (positions 2 to 4)
- `asaka` (positions 2 to 6)

- `aka` (positions 4 to 6)

(3) Show an efficient algorithm to answer query aXa, using a suffix trie. Here, the answers can be the start and end positions of the given string.

This algorithm solves the general query `c1 X c2`, which finds all substrings starting with character `c1` and ending with character `c2`. The query `aXa` is a special case where `c1` and `c2` are the same.

The algorithm leverages a suffix trie to efficiently perform the first part of the task: identifying all possible starting positions.

Algorithm Strategy

The algorithm is divided into two main phases:

1. **Find Start Positions:** The suffix trie groups all suffixes by their prefixes. This means all suffixes of the string that start with `c1` will share a common path from the root. We can exploit this to find all occurrences of `c1` in a single traversal of the relevant subtree.
2. **Find End Positions:** For each starting position `i` found in the first phase, we perform a linear scan through the rest of the string to find all positions `j` (where `j > i`) that contain the character `c2`.

```
// Assume the Suffix Trie for the string S is pre-built.

FUNCTION Find_Pattern_Matches(S, trie, c1, c2):
    // Inputs:
    //   S: The original string of length n.
    //   trie: The pre-built suffix trie for S.
    //   c1: The starting character of the pattern.
    //   c2: The ending character of the pattern.
    // Output:
    //   A list of (start, end) position pairs for all matching substrings.

    // --- Phase 1: Find Start Positions using the Suffix Trie ---

    // 1. Traverse from the root to the node corresponding to the first character.
    Node c1_node = trie.root.getChild(c1)

    // 2. If no such path exists, there are no matches.
    IF c1_node is NULL:
        RETURN empty list

    // 3. Collect all start indices from the leaves of the subtree rooted at
c1_node.
    //    This gives us the 0-indexed positions of all occurrences of c1 in S.
    List<Integer> start_positions = trie.collectAllLeafIndicesFrom(c1_node)

    // --- Phase 2: Find End Positions by Scanning ---

    // 4. Initialize a list to store the results.
    List<Pair<Integer, Integer>> results = new empty list
```

```
    // 5. For each valid starting position...
    FOR EACH start_pos IN start_positions:
        // 6. ...scan the rest of the string to find valid ending positions.
        FOR j FROM (start_pos + 1) TO (n - 1):
            IF S[j] == c2:
                // 7. A match is found. Add the (start, end) pair to the results.
                results.add(Pair(start_pos, j))

    // 8. Return the complete list of matching positions.
    RETURN results
```

(4) Suppose the input string has length *n*, and *d* distinct characters are used. What is the time complexity of your algorithm in (3)?

The time complexity of the algorithm is determined by two main operations during the query phase:

1. Finding Start Positions: This involves traversing a subtree within the suffix trie. In the worst case for a standard (uncompressed) trie, the size of this subtree can be proportional to n^2, leading to a complexity of $O(n^2)$ for this step.
2. Finding End Positions: For each of the `k` start positions found, the algorithm scans the remainder of the string. This nested loop structure results in a complexity of $O(k \cdot n)$.

In the worst-case scenario, `k` can be as large as `n` (e.g., for the string "aaaaa..."). Therefore, the total time complexity is dominated by these operations, resulting in $O(n^2 + n \cdot n) = O(n^2)$. The number of distinct characters, `d`, does not affect the asymptotic time complexity.