CS 344: Design and Analysis of Computer Algorithms

Instructor: Kangning Wang

Homework ∞: Due on **May 7, 2025**

Acknowledgment: Our team members are Alexander Lin (al1655), Pranav Tikkawar (pt422), and Ivan Zheng (iz72) .

Problem 1

We have a rooted tree with n vertices. The height of the tree is h. We want to create a data structure that can quickly find the *lowest common ancestor* of two queried vertices.

We say that a vertex u is an ancestor of a vertex v if u is on the simple path from v to the root (including v and the root). The lowest common ancestor of two vertices is the deepest vertex that is an ancestor of both of the two vertices.

We can compute the depth of all the vertices in $O(n)$ time using DFS or BFS. In addition, we want to record $f(v, k)$ for each vertex v and each $k \in \{0, 1, \ldots, \lfloor \log_2 h \rfloor\}$, where $f(v, k)$ is the 2^k-th ancestor of the vertex v. (The 0-th ancestor of v is v itself, and the $(k+1)$-st ancestor of v is the parent of the k-th ancestor of v.)

Explain how to compute all of $f(v, k)$ in $O(n \log h)$ time and how to use this information (along with the depths of all vertices) to answer any query in $O(\log h)$ time.

Solution. To compute all of $f(v, k)$ in $O(n \log h)$ time, we start by defining:

- $f(v, k)$: The 2^k-th ancestor of v.
  - $f(v, 0)$ = parent of v (or v itself if you define 0-th ancestor as self)
  - $f(v, k)$ = $f(f(v, k-1), k-1)$ for $k > 0$

Now for the depth and parent computation:

- Do a DFS or BFS from the root
- Record parent[v] for each vertex v
- Record depth[v]
- This takes $O(n)$ time

Next is the DP table for $f(v, k)$:

- For each v, set $f(v, 0)$ = parent[v]

- For k = 1 to $\lfloor \log_2 h \rfloor$:
  - For each vertex v: $f(v, k) = f(f(v, k-1), k-1)$
  - Each level doubles the distance moved up the tree
- There are $O(logh)$ levels (since max k is $\lfloor \log_2 h \rfloor$)
- For each level, you process all n vertices
- Total time: $O(nlogh)$

To use this to answer queries in $O(logh)$ time given two vertices u and v:

1. Equalize depths:
   - If depth[u] $<$ depth[v], swap u and v
   - For k descending from $\lfloor \log_2 h \rfloor$ to 0:
     - If depth[u] - $2^k \geq$ depth[v], set u = $f(u, k)$
   - Now u and v are at the same depth
2. Binary lifting to find LCA:
   - If u = v, return u (they are the same node)
   - For k descending from $\lfloor \log_2 h \rfloor$ to 0:
     - If $f(u, k) \neq f(v, k)$, set u = $f(u, k), v = f(v, k)$
   - Now u and v are children of the LCA
   - Return parent[u] (or $f(u, 0)$)

Each of the above steps loops over k = $O(logh)$, so query time is $O(logh)$.

Problem 2

We have an ordered sequence of n operations and a robot that performs these operations. There are two types of operations:

- Forward: Move one step in the direction it faces.
- Turn: Turn $180°$ (and face the opposite direction).

We are given a parameter k, and we can choose to modify at most k operations in the sequence (Forward to Turn, or Turn to Forward). What is the farthest place (farthest in terms of the distance to the starting point) that the robot can arrive at after performing the entire modified operation sequence? Give a polynomial-time algorithm to solve this problem.

Solution. Let $dp[i][j][d]$ = maximum position robot can reach after processing the first i operations, using j changes, and facing direction d.

- i ranges from 0 to n
- j ranges from o to k
- d is 0 for facing right and 1 for facing left

Initialize:

- $dp[0][0][0] = 0$ (starting at position 0, facing right, 0 changes)
- All other $dp[0][*][*] = -\infty$ (invalid states)

At each operation i, for each number of changes $j \leq k$, and for each direction d:

1. Don't modify the current operation
    - If it's an F, move one step in direction d
    - If it's a T, flip direction
2. Modify the current operation
    - This increases change count $j \to j + 1$
    - Change F to T or T to F and apply the corresponding effect

Update $dp[i+1][j][new\_dir]$ and $dp[i+1][j+1][new\_dir]$ for each possible case.

The furthest distance is $\max_{\substack{j \leq k \\ d \in \{0,1\}}} |\mathrm{dp}[n][j][d]|$. The total DP table size is $O(nk)$ and each state takes constant time to compute.

Problem 3

Consider the following linear-time algorithm that finds a longest simple path on any tree. We assume that the edges do not have weights for simplicity, but the algorithm also works if the edges have positive weights.

- Pick an arbitrary vertex u.
- Use DFS or BFS to find a vertex v that is the farthest away from u.
- Use DFS or BFS to find a vertex w that is the farthest away from v.
- Output the simple path from v to w.

Prove the correctness of this algorithm.

Solution. We define:

- diameter(T) as the longest simple path in the tree T
- v as the node farthest from the arbitrary node u
- w as the node farthest from v
- P = path(v,w) as the path returned by the algorithm

We want to show length(P) = diameter(T)

Lemmas:

1. The longest path in a tree is between two leaves.
    - Any simple path in a tree can be extended only by moving toward a leaf.
    - Therefore, the diameter lies between two leaf nodes.
2. The node v, being farthest from u, must be one end of the diameter.
    - Let x—y be the endpoints of the true diameter.
    - Suppose v is not one of them.
    - Then, the path from x to y must pass through u, which contradicts v being farther from u than both x and y (since paths in trees are unique and acyclic).
    - Therefore, the node v is an endpoint of the diameter.
3. Doing BFS/DFS from v finds the other endpoint w of the diameter.
    - Since v is one end of the longest path, the farthest node from v must be the other end w.
    - BFS/DFS guarantees that it finds the farthest node in terms of number of edges (or distance if weighted), so it finds the other end of the diameter.

In conclusion, the path from v to w is:

- A simple path (since the tree has no cycles)
- Of maximum possible length (since it connects two endpoints of the diameter)
- Therefore, it must be the diameter of the tree.

Thus, the algorithm is correct.

Problem 4

Give a polynomial-time algorithm to find all strongly connected components in a given directed graph $G = (V, E)$. Two vertices u and v are in the same strongly connected components if and only if there is a walk from u to v and there is a walk from v to u.

In fact, this problem can be solved in $O(|V| + |E|)$ time. As a voluntary challenge, look up and understand such an algorithm.

Solution. We can use Kosaraju's algorithm to find all SCCs in a directed graph G = (V,E):

1. DFS on G and record finishing order
    - Run DFS on G.
    - When each node finishes (all its descendants are explored), push it to a stack (or record post-order).
    - This gives you an ordering where sink components finish first.
2. Transpose the graph G^T
    - Reverse all edges: For each edge $(u, v) \in E$, add (v,u) to G^T.
    - Time complexity for this step is $O(|E|)$.
3. DFS on G^T in reverse finishing order
    - Pop vertices from the stack.
    - For each unvisited vertex, perform DFS and mark all reachable vertices. They form one SCC.
    - Continue until all vertices are visited.

- Each DFS takes $O(|V| + |E|)$ time.
- Reversing the graph also takes $O(|E|)$ time.
- So the total is $O(|V| + |E|) + O(|E|) + O(|V| + |E|) = O(|V| + |E|)$ time.

Problem 5

Give a polynomial-time algorithm to solve the 2SAT problem. The 2SAT problem is similar to the 3SAT problem except that every clause of it has exactly 2 literals. (Hint: The 2SAT problem is related to strongly connected components.)

Solution. Unlike 3SAT, which is NP-complete, 2SAT can be solved in polynomial time. We can do so using implication graphs and SCCs.

For each clause $(a \vee b)$, interpret it as two implications:

- $(\neg a \implies b)$
- $(\neg b \implies a)$

This forms a directed graph with 2n vertices (one for each literal and its negation). Then, we can use Kosaraju's algorithm to find SCCs in the implication graph, which takes $O(n + m)$ time.

Then, for every variable x, check if x and $\neg x$ are in the same SCC $\implies$ unsatisfiable. This is because it implies both $x \implies \neg x$ and $\neg x \implies x$, which forces both to be true simultaneously, which is a contradiction.

For the time complexity:

- Building implication graph takes $O(n + m)$
- SCC decompositiontakes $O(n + m)$
- And checking each variable takes $O(n)$

Overall, it takes $O(n + m)$ time.