Assignment 6

CSCI 270

Paul Kim | pbkim@usc.edu | 1723717002

Question 1

(1) [10 points]
In CSCI 104, you learned about balanced search trees. Recall that you have n items with keys (let's say numbers) $k_1 < k_2 < \cdots < k_n$. You want to build a binary tree which will later give you low search cost. The search cost is determined by the number of comparisons you need to make for a query: starting from the root, you compare the query q to the key. If it matches, you are done; otherwise, you continue in the left or right subtree. If it takes you c comparisons to find q in the tree, then this corresponds to a cost of c. The point of making the search trees balanced was that it ensured that the height was at most $O(\log n)$, so the cost for *every* query q was at most $O(\log n)$.

Doing so is a very good idea when all queries are roughly equally frequent, or when you have no idea which queries will be frequent. But if you knew that some keys are searched for much more often, you would want to build your tree in such a way that those keys are close to the root (and thus cheap to search for), even if it means making your tree unbalanced and other (rare) queries more expensive. Here, you should give an algorithm to optimally design a search tree, assuming that you know the frequencies exactly ahead of time.[1]

For each of the n keys, you will be given a relative frequency $p_k \geq 0$, such that $\sum_k p_k = 1$. You want to build a binary tree in which each node contains exactly one key k, and each key is in exactly one node. We consider the root level 1, its children level 2, and so on. For a tree T, we let $\ell_T(k)$ denote the level at which the key k occurs. Then, the (expected) cost for T is $\sum_k p_k \cdot \ell_T(k)$, because a p_k fraction of the time, the query q will be for k, and in that case, answering it involves comparing q to $\ell_T(k)$ keys in the nodes of the tree.

Give and analyze (prove correct and analyze running time) a polynomial-time algorithm for computing a search tree T of minimum expected query cost.

Answer:

*AVL/Red Black Trees are useful when all queries are roughly equally frequent and you have no other information available.

*Given in this problem is that we know that some keys are searched for more often than others, and we know the exact frequencies ahead of time.

*We keep in mind that the tree is still a BST (we want it to be a BST), and is ultimately sorted by the key rather than the frequency. Therefore, we cannot simply place the nodes with the largest

frequencies closest to the root, because this may violate the BST property. We have to dynamically resolve a solution that tries all combinations of a sub tree to find a minimal cost, and use the sub-solution to achieve a larger tree.

*Piazza @392: We can assume that the N keys are given in a sorted array. For each of the N keys, we are given a relative frequency such that the sum of all these frequencies is equal to 1. The binary search tree contains unique keys, and each key is in exactly one node.

***Goal: Give an algorithm to optimally design a search tree assuming that we know the frequencies ahead of time, to compute a search tree T of minimum expected query cost.**

Observation: The maximum query time occurs when the most frequent elements are towards the bottom. Therefore, we want to put the highest frequent elements on the top of the tree to reduce total query time, while also maintaining the BST property.

Intuitively, we want to calculate the ordering that yields the minimum cost of a sub tree, and use those solutions to create and build upon to the larger tree.

Dynamic Programming:

An optimal BST is a BST that minimizes the expected query time which is equal to:

$$C(T) = \sum_{i=1}^{N} P_i(Level(K_i)) \tag{1}$$

*Where there are N number of keys (nodes), K = Keys(1,2...,n), P = relative frequencies of the Keys(1,2...n).

We define OPT(i,j) = The minimum expected total query cost of the optimal search tree T.

*Define X_{ij} as the summation of the relative frequencies from key(i) to key(j) at the given level.

*A(i, r-1) represents the total cost of the left sub-tree, and A(r+1, j) represents the total cost of the right sub-tree.

Recurrence Relation: OPT(i, j) = $min_{i<=r<=j}$(A(i, r-1)+A(r+1, j) + X_{ij}), if i < j and where r is the root.

Formal Algorithm:

Let A[n][n] be the array to store solution of minimum query. // N nodes.

Let Root[n][n] be the array to store ordering of optimal tree. // N nodes.

if (n == 0) // Base Case

    RETURN 0 and EMPTY Root;

// For graphs with only 1 element, the frequency of the single element is 1 given by the problem statement (intuitively the root node).

if (n == 1) // Another Base Case

    RETURN 1 and the root itself;

// Array [i][j] means all keys from i...j in a sorted order.

```
for (int i = 0; i < n; i++) // Initializing Values.
    A[i][i] = P[i];
for size from 2 to n // Size of Sequences (Start from 2, since connections are now >= 2)
    for i from 0 to (n-size+1) // Starting Points of Sequence
        int j = size + i - 1
        A[i][j] = 99999; // An assigned arbitrary max value
        int cost;
        for root from i to j // Potential Roots from i to j (we must loop and pick optimal)
            if (root > i) // Out of bounds case
                cost += A[i][root-1]
            if (root < j) // Out of bounds case
                cost += A[root+1][j]
            cost += X_itoj // Recurrence (Edge Cases Accounted For)
            // Where X_itoj is the sum of the relative frequencies P[i] to P[j] (As-
            sume we made a function that did this)
            if cost < A[i][j] // We search for the cheapest cost
                A[i][j] = cost;
                root[i][j] = root; // When this root is used, we achieve the
                cheapest cost... store
```

RETURN A[0][n-1] and Root[0,n-1]. // Where A returns the minimum query cost of the tree, and Root at [0,n-1] returns the root of the tree that yields the minimum query cost.

Runtime: O(N^3) where we have three nested for loops. We can assume that X_{itoj} is precomputed and takes constant time. Each of these for loops take at most N (number of keys) values. Therefore, the worst case time is N^3.

Correctness Statement (Inductive Hypothesis): A[i][j] = OPT(i,j) for all i,j. Then the final answer at A[0][n-1] = OPT(0,n-1) is the optimal solution that returns the minimum expected query cost of a BST. Subsequently, the resulting Root array at the same index also yields the root of the optimal tree.

Proof of Correctness: Induction

We define OPT(i,j) = The minimum expected total query cost of the optimal search tree T.

Prove by induction on i.

Base Case: If there are 0 keys, we return 0 and an empty tree. If there is only 1 element, then we return 1 (the expected query cost summation is 1) and we return the root of the BST (this must be the root, if there is only one element).

Inductive Step: Consider i+j > 0

We express our recurrence as OPT(i, j) = $min_{i<=r<=j}$(A(i, r-1)+A(r+1, j) + X_{ij}), if i < j and we explicitly account for a few base cases (where we check if the indexes reach out of bounds). Here we use the optimal solution from the left sub tree given by A(i, r-1) and the optimal solution from the right sub tree given by A(r+1, j) to continue to build the sum of query costs of the total solution to find one that yields the cheapest cost. Further here, we must prove that the substructure is optimal for a formal full proof.

Let T be an optimal BST for a number of keys with given relative frequencies. Suppose T has a root node R.

For the sake of contradiction, let us assume that the left sub tree T(l) is not optimal for 1,2,...R-1.

Let us obtain T* from T by cutting out this non-optimal T(l) sub tree, and pasting in the optimal T*(l) in for T(l). *Something similar we did for edges when proving Dijkstra's (edge exchanges). We know that an optimal BST is a BST that minimizes the expected query time which is equal to:

$$C(T) = \sum_{i=1}^{N} P_i(Level(K_i)) + 1 \quad (2)$$

Similarly if we were to compute the expected query time of T*, we would see that if we were to do the cut and paste that we performed earlier, then C(T*) < C(T) which contradicts the optimality of T as tree T has a higher cost than T*. Therefore, T(l) (the sub-tree) must have been an optimal solution in the first place. As we proved the optimality of the sub problem, by induction, we can combine the solutions of these sub problems to calculate the entire optimal tree. *This proof is exactly the same/synonymous to the right side*

To prove our recurrence statement, what we are doing is creating sub-problems, and from those sub-problems, trying different root values that yield the most cheapest query cost; that is why we loop and check every root value and take the one that is cheapest.

A[i][j] = $Min_{i<=r<=j}$(A[i,r-1] + A[r+1,j] + X_{itoj}, when i < j and to take into account out of bounds errors. // Algorithm Assignment

= $Min_{i<=r<=j}$(OPT[i,r-1] + OPT[r+1,j] + X_{itoj} // By the Inductive Hypothesis

= OPT(i,j) // What we derived from OPT

Dynamic Programming Proof Completed.

Question 2

(2) [3+10=13 points]

As a child, you might have had nice neighbors who (1) owned fruit trees, and (2) let you climb the trees and pick some of the fruit for yourself. Knowing the fruit eating prowess of children, such neighbors would have had to be careful to avoid you just completely emptying their tree. So they might have given you a time limit on how long you are allowed to be in the tree. In turn, that would create an optimization problem for you: how to get the most fruit in the allotted time. Here, we will solve this problem.

We model the fruit tree as a binary tree T with n nodes. For each node v of the tree, you are given the amount of fruit $f_v \geq 0$ at that place. The neighbor's total time limit is $\tau \geq 0$. We will assume that climbing along one branch/edge of the tree always takes exactly one unit of time (in either direction). You start at a designated root node r, and must finish at r again after at most τ time steps. Your goal is to find a climbing route maximizing the total amount of fruit you collect. Specifically, if your route visits the set S of nodes, then you get $\sum_{v \in S} f_v$ units of fruit.

(a) Prove that if $\tau \geq 2(n-1)$, you can completely clear the tree, i.e., get all the fruit.

(b) Give (and analyze) a polynomial-time algorithm for maximizing the amount of fruit collected within τ units of time. Your algorithm should return the amount of fruit you collect, as well as the set S of vertices you visit.

Answer: A.) ***Under the assumption that picking a fruit takes 0 time (Piazza).**

Tau represents the neighbor's total time limit always greater than or equal to 0. If the time limit is greater than or equal to 2(n-1), then it must be true that the child/individual that will climb up the tree, will be able to completely clear the tree in this time span. This is because, 2(n-1) represents the maximum weight value of a binary search tree with N nodes. We are also given that each weight along an edge takes exactly one unit of time in either direction to traverse (each traversal costs, 1 unit of time). That is why, if given a Tau value greater than or equal to the maximum weight of the tree, and since every traversal of the path takes exactly one unit of time, then you will have enough time to traverse through every path in the allocated time limit.

Proof of Weight of Binary Search Tree: We can assume that each node has a base weight of 1, plus 1 weight for each child (maximum of 2 children since definition of binary tree). The implication for this, is that each node added to the tree become a child of another node, and will contribute 1 weight to its parents. With N nodes, this means there is a total addition of N base weight (from all the individual nodes) and an addition of N-1 total extra weight (to the parent nodes ... and -1 because the root doesn't count).

To simply allocate values:

A node with no children has a weight value of 1.

A node with 1 child, has a weight value of 2.

A node with 2 children, has a weight value of 3.

* We can simply look at it this way as again, each edge traversal costs 1 unit of time.

We can also reference the Handshaking Lemma (CSCI 170) which tell us that the number of edges

in a graph is one less than the number of vertices. If there are N vertices, then there are N-1 edges. It will take at most N-1 times to traverse through all edges given that it takes 1 unit of time to climb. Subsequently, it will take N-1 times to get back to the starting node which is equivalent to 2*(N-1).

To further lastly build more intuition on this problem statement / path cost for part b, a traversal to a node, and back to where you began will take 2 total steps; one step going forward, and 1 step going backward. We can extend this further by taking a look at a linear tree. If there are N nodes, it will take N-1 steps to reach the last node of the tree, and N-1 nodes to go back to where you began.

To sum, if given a Tau value is >= 2(N-1) time, you have enough time to traverse through all the edges and get all the fruit, under the assumption that picking a fruit costs 0 time (Piazza).

B.) *** Optimization Problem: Get the most fruit in the allotted time.**

***** I will denote Tau as T in this problem.

***** Fruit tree is modeled as a binary tree T with n nodes. At each node, we are given an amount of fruit F(v) >= 0. We are given a total time limit T >= 0, and we assume that climbing along one edge takes exactly one unit of time in either direction. We start at a root node r, and we must finish at the same root r again after T steps.

***** Give an algorithm for maximizing the amount of fruit collected within T units of time. Return the amount of fruit you collect, as well as the set S of vertices you visit.

// Similar to Weighted Knapsack Proof

Dynamic Programming:

We define OPT(i,T) = The maximum of fruits collected starting at the ith node (given root), given T time

Recurrence Relation: OPT(i,T) = max(OPT(i.left, T-2), OPT(i.right, T-2), $max_{T' in (1<=T-4)}$ (OPT(i.left, T'), OPT(i.right, T-T'-4).

Proof: We indicate a deduction of 2 from the total time (T-2) because it takes one unit of time to get to one node, and one unit of time to get back from where you started, for a total of 2 units of time.

*In our OPT, we have three parameters. The first parameter, is if we decide to traverse to the left child given it's fruit value. The second parameter, is if we decide to traverse to the right child given it's fruit value, Lastly, the third parameter is the split amount of time between the left and right sub child's (sub-tree); if this is the case, it must be true that if we split both, then there is a T-4 deduction rather than T-2.

Pseudo Code (Tried my best to make formal):

Create a Set S value that will hold all vertices visited

Create a Sum; // for T >= 2(N-1)

Create a 2D Array A[N][T] where N is the of nodes, and T is the Time Given

// Proven that we are able to collect all the fruits given this condition.

if (T >= 2(N-1)) // Case where we are given ample time (puts bound on runtime of T to N)

-> Traverse through every vertex, add to set S, collect/sum it's fruit value and add to sum (something like BFS and BFS runtime).

RETURN all vertices from set S, and sum of fruits (the total of the entire tree).

if (T == 0) // Base Case

RETURN first node, and it's value (if it exists). (can still pick the fruit you are already on at T = 0).

// Initialize the value of the 2D array at T = 0. (Indicate getting the initial fruit)

for i to N // Initialize

A[i][0] = fruit at i at time 0.

// We want to fill up the 2D array in an order such that any time we query a smaller sub problem, it will have been calculated already. When we do a recursive call, we can see that T is always smaller. However, the outer loop should be looping through the time parameter, and the inner loop through the nodes. (Similar to Knapsack)

for time from 1 to T; // Time Parameter

for node from 1 to N; // Iterate through all nodes (it is either starting from 0 or 1, depending on how we implement)

// Cases (Psuedo-Code... something like this)

if (node has no children);

add node to S;

break;

else if (node has 1 children)

if (node.left == true)

add left node to S;

OPT(node.left, T-2)

else

add right node to S;

OPT(node.right, T-2)

else (node has 2 children)

add node to S;

// We follow the recurrence relation:

OPT(node,T) = max(OPT(node.left, T-2), OPT(node.right, T-2), $max_{T' in (1<=T-4)}$ (OPT(node.left, T'), OPT(node.right, T-T'-4).

RETURN Set S and A[0][T]; // Set S keeps track of all vertices that we explored, and A[0][T] keeps track of the total fruit collected.

Runtime: Pseudo-Polynomial is O(n*T^2). However, since we have the conditional statement above where we check if T >= 2(N-1), what this essentially does, is it puts a bound on the maximum running time since we know that T <= N. Since we know that T is bounded by N, we can assign a worst-case running time to this algorithm as O(N^3);

// Running time is O(N*T^2) but the T is upper bounded by N, so it becomes O(N^3),

Proof that Algorithm Returns Correct Set S;

*Note for conciseness, we would also like something in our algorithm to make sure the node doesn't already exist in the set S so we don't have any duplicates.

If we claim our algorithm returns the correct Set S of all the visited nodes, then it must be true that we maximized the amount of fruit we collected. What our OPT/Recurrence relation is doing is exactly that. We want to traverse/make optimal solutions for the sub trees by traversing through the paths that yield the maximum fruits. While doing so, we also check how many children the current node has, and if edges are traversable.

Main intuition is that the algorithm only explores/follows the path that will yield the most fruits in T time. So every node that we added to set S thus far, is the optimal node.

If the current node has 0 children, it must be so that a left and right sub tree does not exist. Therefore, we can break out of the loop, and add that node to the set.

If the current node has 1 children, then we optimize by following the sub tree on the side which the children exist, and following the recurrence criteria.

Lastly, if the current node has two children, then we follow the recurrence relation that optimizes the number of fruit collected given T time.

Proof that Our Algorithm Returns the Amount of Fruit we Collect (that OPT value is correct).

Correctness Statement; A[N][T] = OPT(N,T) for all N, T. Then the final solution at A[0][T] = OPT(0,T) is the sum that yields the highest number of fruit collected given T time.

Proof: We indicate a deduction of 2 from the total time (T-2) because it takes one unit of time to get to one node, and one unit of time to get back from where you started, for a total of 2 units of time. Therefore, this allows us to make sure that we can always get back to the starting root. For the 3rd parameter of the recurrence, since we are splitting time between both left and right sub trees, we indicate (T-4) to account for the back and forth costs x 2.

Proof By Induction on I.

We prove that all the earlier arrays are all correct as our algorithm fills up the 2D array in an order such that any time we query a smaller sub problem, it will have been calculated already. When we do a recursive call, we can see that T is always smaller. The outer loop is looping through the time parameter, and the inner loop through the nodes.

Since we calculate the smaller sub problems first, then the induction part works.

Base Case: T = 0, we can still pick the fruit that you are already on (if it exists), therefore, the set S will contain that root node, and the amount of fruit at the node.

We also have a case where if T >= 2(N-1) then it is possible to collect all the fruits and reach all

the nodes of the binary tree. We can use an algorithm like BFS to search/iterate through the tree to collect the set S and fruit value.

Inductive Step:

A[N][T] = max(A(i.left, T-2), A(i.right, T-2), $max_{T'in(1<=T-4)}$ (A(i.left, T'), A(i.right, T-T'-4). // Algorithm Assignment

= max(OPT(i.left, T-2), OPT(i.right, T-2), $max_{T'in(1<=T-4)}$ (OPT(i.left, T'), OPT(i.right, T-T'-4). // By Inductive Hypothesis

= OPT(N, T) // What we derived from OPT

// Induction step uses the justification of recurrence relation.

Dynamic Programming Proof Completed.