Gray Houston
ghousto
0026483532

Project 4 Analysis Questions

For all questions, the variable '*n*' refers to the number of vertices and the variable '*m*' refers to the number of edges.

1. The asymptotic performance of my recursive tree height function is O(*n*), where *n* is the number of nodes in the tree. The algorithm basically works by finding the maximum height of the current children and returning that height plus 1, while watching for the leaf node corner case.

   Because of the tree structure, each node can only be accessed by its parent. Therefore, once a parent node has recursively calculated the height of a child (and the algorithm starts moving back down the recursive stack), that child's height will never be used again and that child node will not ever be accessed again. Proof for this statement:

   > Because the child's height is only used to calculate the parent's height, a child's height is only needed exactly once. Therefore, once a parent's height has been determined, it never needs to be accessed again. Furthermore, because the recursive accessing is essentially a DFS traversal of the tree, each node will be accessed once.

   For this reason, throughout all of the recursive calls, each node will be accessed exactly once. Said another way, there will be *n* node accesses in the algorithm. Therefore, the asymptotic performance of the algorithm will be O(*n*), where *n* is the number of nodes in the tree.

   It should be noted that there will be some overhead because of the recursion. However, as *n* increases and approaches its asymptotic boundary (i.e. as *n* approaches ∞), this recursive overhead becomes insignificant and negligible. Therefore, this overhead shouldn't be considered for the asymptotic performance.

2. The asymptotic performance of my non-recursive tree height function is O(*n*), where *n* is the number of nodes in the tree.

   The algorithm uses as BFS approach to traversing the tree. It basically works by adding the children of all the elements of a certain level (i.e. all vertices with the same height), beginning with the root's children, to the queue, then removing the current level's nodes from the queue. Each time a new level is reached (i.e. the queue isn't empty), the height counter increases. Thereby, the algorithm traverses the tree by moving from one height level to the next.

Excluding the root, each node is accessed twice: first when it is added to the queue, and second when it is removed from the queue (when its children are added to the queue). Because the root is never added to the queue, it is only accessed once. Therefore, the running time for the algorithm is $O(2n - 1)$. Therefore, the asymptotic performance (as n approaches ∞) for my algorithm is $O(n)$, where n is the number of vertices in the tree.

3. As stated above, the recursive tree height function essentially implements a Depth-First Search (DFS). The algorithm recursively moves down the levels of the tree (from parent to child) until it hits a leaf. When a leaf is hit, it moves up one level to the parent, and then recursively tries to follow other children, recursively move down tree height levels until a leaf node is hit.

   Because this algorithm attempts to travel a single path as deep into the tree as possible before moving back up a level and trying a new path, the algorithm is a DFS traversal. Furthermore, because this function operates like a DFS, it traverses all of the nodes in a pre-order traversal (accesses parent first, then child nodes from left to right).

4. For a tree with an n-vertex star shape (a single parent and n-1 children), there are 2 possible pebbling placements that could maximize profit: pebble the root and none of the children, or pebble all of the n-1 children and not the root. All the other possibilities would be different combinations of the root not being pebbled with one of the $(n\text{-}1)! - 1$ permutations where some, but not all, of the child nodes are pebbled. Assuming that profit cannot be negative, the profit from summing of all n-1 children will be greater than the profit of any of the other $(n\text{-}1)! - 1$ permutations, as each permutation will sum the profit of $n - 1 - k$ nodes, where $k > 0$. Said more generally:

   Claim: If the root is not pebbled, the maximum profit pebbling permutation for the child nodes is the permutation where all $n - 1$ children are pebbled

   Assumptions:
   a.) There are a total of $n - 1$ children
   b.) Each individual node's profit is positive
   c.) The root node is not pebbled

   Proof:
   k = number of child nodes not pebbled
   Because of assumption A, $k \geq 0$ & $k \leq (n - 1)$
   p_m = profit of a single, arbitrary child node m
   P = total profit from pebbling
   s = number of child nodes pebbled

   $s = n - k - 1$
   $P = p_0 + p_1 + p_2 + \ldots + p_{n-k-1}$
   $P = p_0 + p_1 + p_2 + \ldots + p_s$
   Therefore, P is the summation of the profits of all pebbled nodes
   Because of assumption B ($p_m > 0$), each additional node pebbled increases P

Therefore, P will achieve a maximum when all child nodes are pebbled (i.e. when the number of nodes pebbled *s* achieves a maximum)

Max(*s*) = $n - 1 \rightarrow k = 0$
Therefore, the pebbling permutation of child nodes with maximum P is the pebbling configuration where all child nodes are pebbled ($k = 0$)

Because of assumption C, this is a valid pebbling
Therefore, the maximum profit pebbling permutation for the child nodes is the permutation where all $n - 1$ children are pebbled

As I have just proven, the maximum profit pebbling when the root is not pebbled is where all of the child nodes are pebbled. Therefore, we need to determine whether to pebble to root node or pebble all of the child nodes.

Knowing this fact greatly simplifies the problem. As the child nodes are read in (i.e. as children are added to the tree), add each new child's profit to a sum of all of the profits of child nodes entered so far. If, after a node is entered, the sum of the parsed child nodes' profits exceeds the root's profit, you will immediately know that all of the children should be pebbled and the root should not. At this point, you do not need to read in any more nodes and you can immediately return the correct pebbling array.

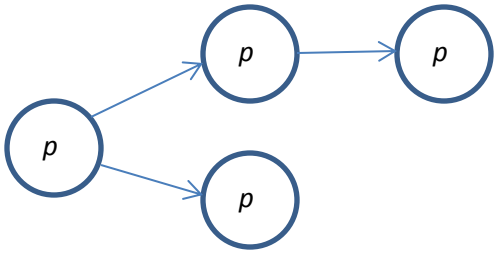
If, after all of the child nodes have been read in, the root's profit is greater than the sum of all of the child nodes' profits, then you know that the proper scheme is that the root should be pebbled and none of the children should be.

Because $n - 1$ elements will be accessed in the worst case to sum all the child nodes and the root node must be accessed to determine if the root has the largest profit, there will be a total of *n* nodes iterated/traversed. Therefore, the worst case running time for this algorithm will be O(*n*), where *n* is the number of nodes in the tree.

5. The claim that any given tree always has a unique optimal pebbling is **false**. I will use a Proof by Contradiction to show that the optimal pebbling is not always unique for all trees.

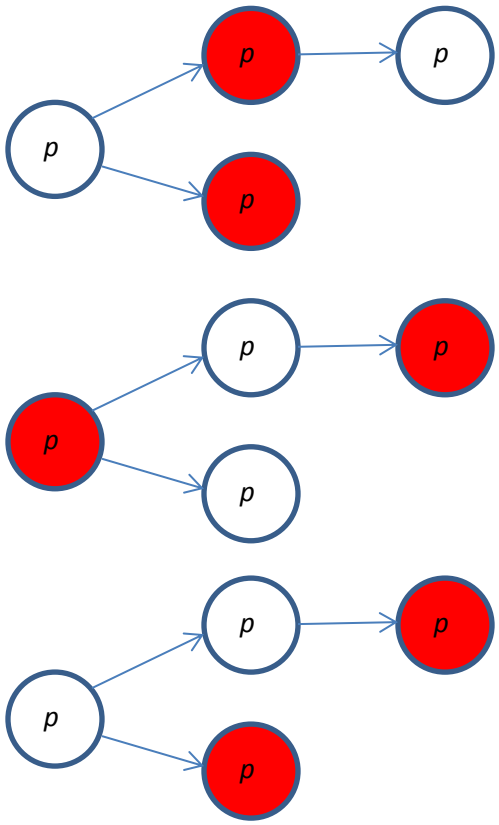
   Claim: the optimal pebbling is not always unique for all trees
   Assume: the optimal pebbling is always unique

   For example, take a tree where $n = 4$ and each node has the same profit *p*



Because the tree has a size of $n = 4$, the maximum number of nodes that could be pebbled is $n / 2 = 2$
Therefore, this tree has a maximum profit of $P = 2p$

Because of our assumption, there should only be 1 pebbling where $P = 2p$
However, there is more than one possible pebbling scheme that could produce this value



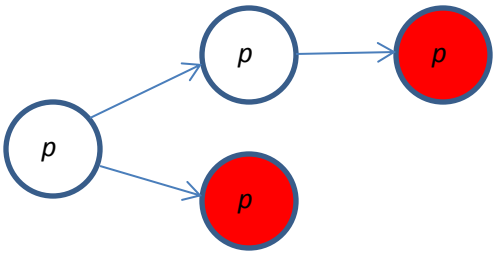
• • •

The fact that more than 1 tree produces the optimal profit creates a contradiction
This contradiction proves that our assumption is wrong
Therefore, I conclude that an optimal pebbling is not unique for all trees

6. The claim that for an optimal pebbling, every leaf is pebbled is **false**. I will use a Proof by Contradiction and the same example as in question 5 to disprove this claim.

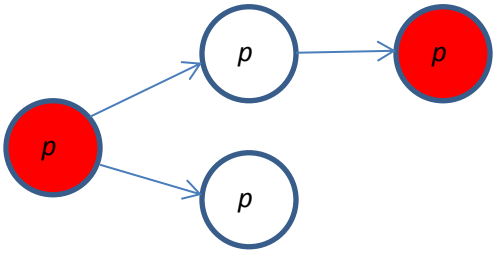
Claim: In a maximum profit pebbling, not every leaf of a tree needs to be pebbled

Assume: In an optimal pebbling, every leaf must be pebbled



Because the tree has a size of $n = 4$, the maximum number of nodes that could be pebbled is $n / 2 = 2$
Therefore, this tree has a maximum profit of $P = 2p$

However, there exists a pebbling with $P = 2p$ in which not every leaf is pebbled



This pebbling produces a contradiction with our assumption
This contradiction proves that our assumption is incorrect

Therefore, in a maximum profit pebbling, not every leaf of the tree needs to be pebbled

7. The asymptotic worst-case performance for my brute-force algorithm is $O(2^n)$ while there are $O(n\ 2^n)$ calculations, where n is the number of vertices in the graph. The nature of the algorithm is brute force because it checks every single permutation in the set of 2^n possible pebbling permutations to determine the maximum profit pebbling. Therefore, this algorithm will run in at least $O(2^n)$ time.

   Also, for each permutation, a Boolean pebbling array must be created that represents the current permutation. With n nodes, the cost of building this array is $O(n)$ computations. Furthermore, because this array building operation occurs for each individual permutation, the $O(n)$ cost is incurred 2^n times. Therefore, the total number of computations in this algorithm is $O(n\ 2^n)$.

   However, $O(n\ 2^n)$ is not the asymptotic performance. As n increases (i.e. as $n \rightarrow \infty$), the value of 2^n will become significantly greater and greater than the value of n. Furthermore, we can determine function will asymptotically grow the fastest using limits and L'Hospital's rule:

Claim: As n approaches infinity, $O(2^n)$ will asymptotically grow faster than $O(n)$

Proof:
Use limits and L'Hospital's Rule

$$\lim_{n \to \infty} \frac{n}{2\text{^n}}$$

$$\lim_{n \to \infty} \frac{1}{2^n * \log(2)} = 0$$

The fact that the limit evaluates to 0 proves that $O(2^n)$ asymptotically grows faster than $O(n)$

Because $O(2^n)$ asymptotically grows faster than $O(n)$, the asymptotic worst-case performance of the brute force algorithm is $O(2^n)$.

8. When running on my machine, the brute force algorithm takes over 5 minutes to run when there are 25 vertices in the graph. Specifically, the run time is 5.22 minutes. When the brute force algorithm runs on a graph containing 26 vertices, it runs in 8.23 minutes. This is a humongous increase in execution time for simply increasing the number of nodes by one. By increasing to $n = 26$, the running time increases by almost exactly three minutes.

9. The asymptotic worst-case performance for the recursive pebbling algorithm is $O(n)$.

   In my implementation, the first step of rooting the tree performs $O(1)$ calculations. Because all n nodes in the graph will have to be traversed at some point later in the algorithm, any random node could be chosen as the root. Therefore, in my implementation, I just randomly select a node as root. The parent-child relationships will be built in a later step.

   In the second step, we simultaneously build a child list for each node and calculate the maximum profit of a given node for if it is pebbled or not pebbled. First, building all of the child lists will require copying at least n object references. Therefore, building all of these lists will asymptotically run in $O(n)$ time.

   When calculating a node's pebbled and non-pebbled maximum profit, the algorithm recursively calculates either the not-pebbled or the maximum of pebbled and not-pebbled, depending on the function. Therefore, a maximum profit is calculated at least twice for all n nodes. However, this algorithm makes use of dynamic programming by storing the result of the pebbled profit and not-pebbled profit into arrays. Because of this dynamic programming, a maximum profit for each n node is calculated exactly twice. For this reason, the number of computations is proportional to $O(2n)$ and all of the asymptotic worst-case for this operation is $O(n)$.

In the third step, the algorithm determines the pebbling for each node based on pebbling rules and looking if the profit would be larger if the node was pebbled or not pebbled. Because all n nodes will be accessed, this operation will asymptotically run in $O(n)$ time.

Because none of the steps asymptotically run slower than $O(n)$, the asymptotic worst-case performance for the entire algorithm is $O(n)$.