CS624: Analysis of Algorithms

Assignment 4 – solution

1. Let T be a rooted tree. T is not necessarily a binary tree. That is, each node may have any finite number of children. We are going to consider a method of visiting each node of the tree. The method works like this:

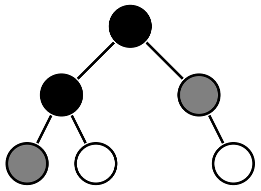
   At Step 1, we visit the root. The root is then marked as "visited".

   At Step n, all nodes that have already been marked "visited" can visit at most one of their children. Of course if all the children of a node have already been visited, there is nothing for that node to do. But other nodes may still do something at that step. (And so in particular, more than one node might be visited in a single step of the process.)

   The question is this: what is the minimum number of steps necessary to visit every node in the tree?

   (a) Draw a picture of a tree in which more than one node is visited at some step of this process, and show why that happens.

   See figure:

   The black nodes are already visited, and the gray nodes are the ones that can be visited at the next stage.

   (b) Show that this problem contains optimal substructure. Please be careful when you do this. You have to state precisely and clearly what the substructure is and show why it is optimal.

   The optimal substructure is that the optimal visitation of a tree contains the optimal visitations of its sub-trees. First you visit a node, then you visit its entire subtree. Therefore, the optimal sub-tree visitation doesn't depend on the root and the optimal visit time is that of the root + the optimal of the subtrees (by standard cut-paste algorithm). Since you can visit more than one node at the same time we should visit the children of the root in descending order of their own optimal visitation times.

   (c) Use that result to write a recursive algorithm to solve this problem.

      i. Visit the node.
      ii. Boundary – if the node is a leaf, finish.
      iii. For each child, calculate its optimal visitation time. Sort the children by the optimal visit time of their sub-trees.
      iv. Visit them in this order.
      v. Obviously, do the visit of children simultaneously while continuing the parent.

(d) Turn that algorithm into a dynamic programming algorithm.

Since there is a lot of overlapping calculations of sub-tree visit time and since there is optimal sub-structure (since the optimal visit time for a sub-tree remains the same whether it is a part of a bigger tree or stand alone), we can think of the visit time of a tree as follows:

i. If it's a leaf, the visit time is 1.

ii. Otherwise, since we visit the children in descending order of their visit time, the visiting time is bound by the finish time of the child that finishes last. This would be the child with the largest visiting time of all the children. So the visiting time of a tree is 1 + the time we finished visiting the last child. If we put the visiting time of the children in an array A sorted by descending order of their visit time, then the visiting time of a tree is $\max A[i] + i$

(e) What is the cost of the algorithm? (By that, I mean, "What is the cost of the algorithm that computes the minimum cost?", not "What is the minimum cost?". In doing this problem, please write as little pseudo-code as possible. I would much rather read something that is clearly explained using ordinary language.

If we have to sort all the children it takes $O(k\ logk)$, where k is the number of children.

2. Determine an LCS of $< 1, 0, 0, 1, 0, 1, 0, 1 >$ and $< 0, 1, 0, 1, 1, 0, 1, 1, 0 >$. Of course you could probably solve this problem easily enough just by looking at it. What I want you to do is to explicitly use the algorithm presented in class. Write out your derivation neatly, including the table.

Here is a table with the path marked. Notice that more than one solution exists. If both left and up arrow were equivalent I chose up, but other paths are possible.

	y_j	1	0	0	1	0	1	0	1
x_i	0	0	0	0	0	0	0	0	0
0	0	↑0	↖1	↖1	↑0	↖1	↑0	↖1	↑0
1	0	↖1	↑1	↑1	↖2	↑1	↖2	↑1	↖2
0	0	↑1	↖2	↖2	↑2	↖3	←3	↖3	←3
1	0	↖1	↑2	↑2	↖3	↑3	↖4	←4	↖4
1	0	↖1	↑2	↑2	↖3	↑3	↖4	↑4	↖5
0	0	↑1	↖2	↖3	↑3	↖4	↑4	↖5	↑5
1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
1	0	↖1	↑2	↑3	↖4	↑4	↖5	↑5	↖6
0	0	↑1	↖2	↖3	↑4	↖5	↑5	↖6	↑6

3. Exercise 3.2 in the Lecture 8 handout (on page 12): The "cut and paste" argument says the following: Since we have independent sub-problems in the sense that the subtree containing nodes $i \ldots j$ where these are consecutive numbers is its own tree, then if this subtree is part of the optimal tree it must be optimal itself. A simple proof by contradiction says that if this weren't the case, we could have cut it out of the tree and plant a better tree instead of it, not affecting the rest of the tree, and get a better result which contradicts our initial assumption that it was optimal.

4. Ex. 3.3: The tree is a BST, so the keys are sorted in dictionary order. The fact that every subtree is a consecutive set of keys follows from the properties of BST and it is not necessarily only particular to this problem.

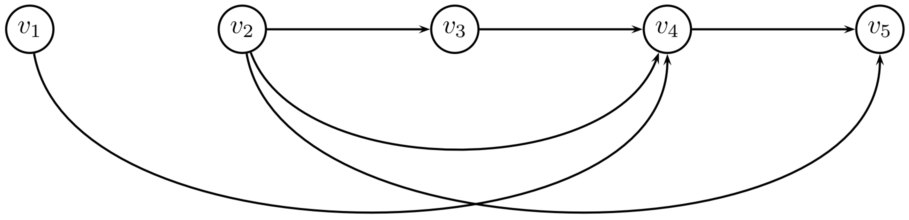
The LCA of i and j, the first and last leaves in the subtree is the root of the subtree (by definition), so it is an ancestor of any other key in the subtree. Now, look at three keys x, y, z

– say that y is x's successor and z's predecessor. We'll show that if x and z are in the tree, y must be too. By HW3, y is either ancestor or descendant of both x and z. Either y is x and z's ancestor, in which case it's their LCA (if it weren't LCA, it wouldn't be both the successor and predecessor of both keys) or the three lie on one path, in which case the uppermost key in the path is the LCA of y and z or x and y. so they all must be in the subtree rooted at the LCA of the entire subtree.

5. DAG:

   (a) The path returned by the algorithm is $v_1 \to v_2 \to v_4 \to v_5$, which is indeed the longest.

   b. Slightly modify the graph in (a) such that this algorithm no longer gives the longest path (**Hint:** It's enough to modify two edges: Delete (v_1, v_2) and add another edge, you have to find out where). Explain briefly why the algorithm above won't work on your example.


$$\begin{tikzcd} v_1 \arrow[rrr, bend right] & v_2 \arrow[r] \arrow[rr, bend right] \arrow[rrr, bend right] & v_3 \arrow[r] & v_4 \arrow[r] & v_5 \end{tikzcd}$$

   The algorithm in (a) will give $v_1 \to v_4 \to v_5$, while the longest path is $v_2 \to v_3 \to v_4 \to v_5$. Other examples exist too. I accepted any example that works.

   c. It is possible to find the longest path using dynamic programming. This is done by calculating, for every vertex v_i, the longest path that ends in v_i. Notice that any vertex v_i extends the longest paths ending at each one of its incoming neighbors by 1 (this is true only in DAGs, of course, due to the fact that paths only go one way, so to speak). Therefore, the length of the longest path ending at v_i is the length of the longest path of all of v_i's predecessors + 1. The algorithm is given below:

      i. For each v_i whose in-degree is 0, set $LP(v_i) = 0$ (this is the longest path ending at v_i).
      ii. For each of the other vertices w_i, in the topological sort order:
         A. set $LP(w_i) = \max_{v_j s.t (v_j, v_i) \in E} LP(v_j) + 1$
      iii. Return $\max_{v_i \in V}(LP(v_i))$

   For each of the vertices in the DAG shown in (a) above, fill in the length of the longest path ending in it.

v_i	$LP(v_i)$
v_1	0
v_2	1
v_3	0
v_4	2
v_5	3

   d. (4%) What is the run time of the algorithm in (c) above as a function of $|V|$ (the number of vertices) and/or $|E|$ (the number of edges)? Explain briefly.

   $O(|V| + |E|)$ Notice that the inner loop in the algorithm above depends solely on the number of edges. Hence, the run time is $O(|V| + |E|)$.

6. MST:

   (a) Given a graph $G = (V, E)$ and MST, if we look at any subgraph $G' = (V', E')$, the portion of the MST that spans G' is optimal with respect to G'. The proof is a simple cut+paste argument - if this were not the case, we could have replaced the set of edges with a cheaper way to span the subgraph, creating an overall better MST, contradicting the initial assumption of a minimum spanning tree.

   (b) First of all, for each cut there must be an MST edge that crosses it, or the tree would be disconnected. Say, by contradiction that for a certain cut, the MST edge that crosses it, e, is not the lightest. Then we can remove e, disconnect the tree and reconnect it using e', the lightest edge that crosses the cut. We can then get a better tree, contradicting the assumption that the original MST is indeed the minimum one.

   (c) This is a simple extension of (b) above. The lightest edge in the tree, $e = (u, v)$ must cross some cut - let's say u on one side, v on the other, and all other vertices anywhere we want.

7. Independent set:

   (a) First notice that there is a bit of a twist here – if we select a node for a maximum independent set, we cannot select any of its children, so the substructure is all the subtrees rooted in the grandchildren. Therefore the optimal substructure is: Given a maximum independent set, then for every subtree (rooted in the grandchildren of selected nodes, not children), the part of the set has to be optimal for that subtree. The proof is a cut and paste.

   (b) Again – be careful here. Not every leaf is a part of every independent set, so don't go that route. Given a maximum independent set S. Given any leaf l – if $l \in S$ we are good. Otherwise, its parent is in the set (if neither the leaf nor its parent are in the set it can't be maximum... think why). In this case we can remove the parent and add l. The new set S' is still independent, because the only neighbor of l is the parent that's now removed, and it is the same size, so still maximum.

   (c) The recursive formula, given a tree T, is as follows:

   $$MS(T) = \begin{cases} Null & \text{if T is empty} \\ T & \text{if T is a leaf} \\ \max\{root(T) + \cup MS(\text{T's grandchildren}), \cup MS(\text{T's children})\} & \text{otherwise} \end{cases}$$

   In other words - if we pick a root, we recursively calculate the max. set of its grandchildren. Otherwise, we calculate the max set of its children, pick the best result.

   However, we said in (b) that every leaf can be a part of a max. set, so we can do it greedily as follows: Pick a leaf at random, add to the set. Delete its parent and all the edges from/to it from consideration (the tree is then disconnedted but it's ok). Continue until done.

8. If a tree is not full – that is, a node has one child, we can remove this node and connect its child to the node's parent, shortening the code(s) by one.

9. If the frequencies are more or less equal, we will get a balanced tree. If we have all 256 characters at more or less equal frequencies, we will get a balanced tree of height 8. In this case we only replace one 8-bit code with another 8-bit code, saving nothing. It also explains (somewhat intuitively) why text files compress a lot better than binary files. Binary files are a lot closer to this scenario of a uniformly distributed sequences of 0's and 1's.