CS624: Analysis of Algorithmns

Assignment 4 – solution

1. Exercise 15.4-1 in the text (page 396). 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.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_j$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$x_i$</td>
<td>0</td>
<td>0</td>
<td>↑0</td>
<td>▼1</td>
<td>▼1</td>
<td>↑0</td>
<td>▼1</td>
<td>↑0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>▼1</td>
<td>↑1</td>
<td>↑1</td>
<td>▼2</td>
<td>▼1</td>
<td>↑1</td>
<td>▼2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>▼2</td>
<td>▼2</td>
<td>↑2</td>
<td>▼3</td>
<td>←3</td>
<td>▼3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>▼1</td>
<td>↑2</td>
<td>↑2</td>
<td>▼3</td>
<td>▼4</td>
<td>←4</td>
<td>▼4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>▼1</td>
<td>↑2</td>
<td>↑2</td>
<td>▼3</td>
<td>▼4</td>
<td>↑4</td>
<td>▼5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>▼2</td>
<td>▼3</td>
<td>↑3</td>
<td>▼4</td>
<td>↑4</td>
<td>▼5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>▼1</td>
<td>↑2</td>
<td>↑3</td>
<td>▼4</td>
<td>↑4</td>
<td>▼5</td>
<td>↑5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>▼1</td>
<td>↑2</td>
<td>↑3</td>
<td>▼4</td>
<td>↑4</td>
<td>▼5</td>
<td>↑5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>↑1</td>
<td>▼2</td>
<td>▼3</td>
<td>↑4</td>
<td>▼5</td>
<td>↑5</td>
<td>▼6</td>
</tr>
</tbody>
</table>

2. Exercise 3.2 in the Lecture 8 handout:

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.

3. 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. Exercise 3.3 in the Lecture 8 handout:

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. Exercise 15-2 (page 405). We need to find the longest palindromic subsequence. Notice that according to the question – the subsequence does not have to be contiguous just like the LCS problem. A palindromic subsequence divides the sequence into two parts (maybe with an extra letter, if the palindrome is odd numbered). So, we can find the longest common subsequence between the string and its reverse. Another way to think about it (same approach, basically) is the following: Given a string \( A[1..n] \), examine \( A[1] \) and \( A[n] \). If they are the same, add them to the palindrome and solve for \( A[2..n-1] \). Otherwise, if \( A[1] \) and \( A[n] \) are not the same, at least one of them is not part of the longest palindrome. Select the maximum result between \( A[2..n] \) and \( A[1..n-1] \). Boundary condition: A string of size 0 or 1 is a palindrome. This is a recursive algorithm and obviously not efficient since there are a lot of overlapping subproblems, so we can use the same formula with DP, saving the results for \( LPS[i..j] \).

6. (longest path in a DAG).

(a) If we follow the code above we choose first the edge \( v_1, v_2 \), then \( v_2, v_4 \) and then \( v_4, v_5 \). This gives us a path of length 3 which is indeed the longest.

(b) In general this will not work, since the nodes are not always “ordered” so neatly as in this graph (which is a directed acyclic graph, DAG). If, for example, we deleted the edge \( v_1, v_2 \) and added an edge \( v_2, v_3 \). Then the algorithm above would give us the path \( v_1, v_4, v_5 \), whereas the path \( v_2, v_3, v_4, v_5 \) is longer.

(c) This is not an efficient algorithm because in general there is an exponential number of paths. A path contains a ordered subset of the nodes, which can be as many as the number of permutations on the nodes...

(d) Since in a DAG all the edges go one way, from smallest to a largest node, we can show an optimal sub-structure such that for any node \( v_i \), its longest path to \( v_n \) is 1 + the longest path from one of its outgoing neighbors to \( v_n \). This is true since if we had a longest path starting at \( v_i \) that does not contain the longest path from one of its outgoing neighbors to \( v_n \) but another path (not the longest) from a neighbor, we could replace it by that longest path from the outgoing neighbor and get a better path (hence a contradiction). The substructures are also overlapping by the definition above – the longest path from \( v_i \) to \( v_n \) contains the longest paths from its outgoing neighbors. In particular, the longest path from \( v_1 \) to \( v_n \) is defined in a similar way, and our boundary condition is that the longest path from \( v_n \) to itself is 0. So we work our way backwards and formulate the DP problem as follows:

\[
path[v_i] = \begin{cases} 
0 & i = n \\
\max_{j \in \{i, j \} \in E} \{ path[v_j] + 1 \} & \forall i < n
\end{cases}
\]

Where we should eventually find \( path[v_1] \).

(e) For each edge we maximize over its outgoing neighbors, therefore we perform \( O(|E|) \) operations, where \( E \) is the set of graph edges.

7. MST:
(a) The optimal substructure here is a standard cut and paste. Given an MST $T$ for a graph $G$, any subtree $T'$ is minimal with respect to the subgraph it represents. Otherwise we could find a smaller MST and attach it to the rest of the tree, getting a better tree which contradicts $T$ being optimal.

(b) Given a cut (any cut) in a graph, suppose the lightest crossing edge $e$ is not in the MST $T$. Adding $e$ to the $T$ creates a cycle (as would be the case for any edge added to a tree). Some other edge $f$ in the cycle must also be a crossing edge, since the MST parts contained completely in one side or the other of the cut don’t have cycles (b/c it’s a tree). We can therefore remove $f$, breaking the cycle and resulting in a new spanning tree, $T'$. But since $e$ is the lightest edge crossing the cut, $\text{weight}(e) < \text{weight}(f)$, therefore $\text{weight}(T') < \text{weight}(T)$ which contradicts the assumption that $T$ is an MST.

(c) this is a corollary on (b). The lightest edge in a graph is always the lightest edge that crosses any cut it appears in.

Notice that in the proof above I assume all edge weights are distinct. If not, there may be an MST that doesn’t contain one of the lightest edges (but it must contain at least one).

8. Independent set of a tree:

(a) This is again a standard cut+paste argument. If we have a max. I-set $S$ for a tree $T$, any subset $S'$ of $S$ must be maximum I-Set w.r.t. the subtree $T'$ it represents, or we could find a bigger subset and replace it.

(b) Notice that I am not asking you to prove that every leaf is in every max. I-set! Obviously this is not true. What you have to show is that for every leaf there is at least one max I-set that contains it (there may be multiple max I-sets in a tree. They all have to be the same size). So, given a leaf $l$ and a max I-set $S$. One of three options exist:

- $S$ contains $l$. We’re done.
- $S$ contains $l$’s parent $m$. We can remove $m$ and add $l$ safely, since $l$ is only connected to $m$, so it’s not connected to anything else in $S$. We get an I-set $S'$ of the same size as $S$, therefore maximal.
- $S$ contains neither $l$ nor $m$. In this case $S$ can’t be optimal because we can safely add $l$ to it ($S$ does not contain $m$, the only node $l$ is connected to) and get a bigger I-set.

This is true for every leaf.

(c) A greedy algorithm is as follows:

- While there is at least one leaf left, add it to $S$.
- remove $l$’s parent from consideration (this means the tree is disconnected, but that’s ok, we’re looking for an I-set). This also means that new leaves may be crated.
- Repeat until no leaves are left. Notice that due to the stage above there may be nodes that are not leaves in the tree but are leaves in the intermediate stage. We treat them as leaves.