CS311: Homework #5

Due on November 11, 2013 at 4:30pm

Professor Lathrop Section 3

Josh Davis

Problem 1

Give algorithms for the following operations.

Part A
Give an algorithm that multiples two degree-1 polynomials with only three multiply operations.

Solution

The normal four multiply operation result obviously (by **FOIL**) is:

$$(ax + b)(cx + d) = acx^2 + (ad + bc) \cdot x + bd$$

Since one of the multiplications is $(a + b)(c + d)$, this gives us the following:

$$\begin{aligned}(a + b)(c + d) &= ac + (ad + bc) + bd \\ (a + b)(c + d) - ac - bd &= (ad + bc)\end{aligned}$$

Substituation gives us:

$$\begin{aligned}(ax + b)(cx + d) &= ac \cdot x^2 + (ad + bc) \cdot x + bd \\ &= ac \cdot x^2 + ((a + b)(c + d) - ac - bd) \cdot x + bd \\ &= m_1 \cdot x^2 + (m_2 - m_1 - m_3) \cdot x + m_3\end{aligned}$$

where $m_1 = ac$, $m_2 = (a + b)(c + d)$ and $m_3 = bd$. This gives us the following algorithm:

```
function MultiplySingleDegreePolynomials($(a, b, c, d)$)
    $m_1 \leftarrow a \cdot c$
    $m_2 \leftarrow (a + b) \cdot (c + d)$
    $m_3 \leftarrow b \cdot d$
    return $(m_1,\ m_2 - m_1 - m_3,\ m_3)$        // Return the tuple of coefficients
end function
```

Algorithm 1: Multiply 1 degree polynomial

Problem 1 continued on next page. . .

Part B

Give a divide-and-conquer algorithm for multiplying two polynomials of degree n. Prove using the master theorem that your algorithm runs in $\Theta(n^{\log_2 3})$. You may assume that $n+1$ is a power of 2.

Solution

```
function MultiplyNDegreePolynomials($p_1, p_2, n_0$)
    if $n_0 = 0$ then
        return $p_1 \cdot p_2$
    end if

    $n \leftarrow n_0/2$
    $a \leftarrow p_1[0 \ldots n-1]$
    $b \leftarrow p_1[n \ldots n_0]$
    $c \leftarrow p_2[0 \ldots n-1]$
    $d \leftarrow p_2[n \ldots n_0]$

    $m_1 \leftarrow$ MultiplyNDegreePolynomials($a, b, n$)
    $m_2 \leftarrow$ MultiplyNDegreePolynomials($c, d, n$)
    $m_3 \leftarrow$ MultiplyNDegreePolynomials($a+b, c+d, n$)        // Sum of the two arrays linearly

    // Shifting and combining runs in $O(n)$ time
    return Shift($m_1, n_0$) + Shift(($m_2 - m_1 - m_3$), $n$) + $m_3$
end function
```

Algorithm 2: Multiply n degree polynomial

Proof of Running Time

Proof. We will prove that the running time of the above algorithm is $\Theta(n^{\log_2 3})$ using the master method.

The recurrence for the above algorithm is:

$$T(n) = \begin{cases} 1 & n = 0 \\ 3 \cdot T(\frac{n}{2}) + O(n) & n > 0 \end{cases}$$

We have $a = 3$, $b = 2$ and $f(n) \in O(n)$. According to the Master Method, this gives us:

$$n^{\log_b a} = n^{\log_2 3} \in \Theta(n^{\log_2 3})$$

This puts us in case 1 of the master method because $f(n) \in O(g(n))$ where $f(n) \in O(n)$ and when $\epsilon \approx 0.58$ because:

$$g(n) = n^{\log_2 3 - \epsilon} = n^{\log_2 3 - \epsilon} = n^1 = n$$

Given the master method, we then can determine the running time of our recurrence to be $T(n) \in \Theta(n^{\log_2 3})$.

□

Problem 2

Give an $O(\log n)$ time algorithm that computes the following function:

MEDIAN-OF-TWO(l_1, l_2)

Input: l_1 and l_2 are two sorted lists of integers. Each list has n elements ($2n$ elements in total) and the value of each element in the lists is unique.
Output: The value of the n^{th} smallest integer in the set of $2n$ integers is l_1 and l_2.

Solution

```
function MID($x, y$)
    return $(y - x)/2 + x$
end function

function MEDIAN-OF-TWO($l_1, l_2$)
    $s_1 \leftarrow 0$
    $s_2 \leftarrow 0$
    $e_1 \leftarrow l_1.length$
    $e_2 \leftarrow l_2.length$
    $mid_1 \leftarrow$ MID($s_1, e_1$)
    $mid_2 \leftarrow$ MID($s_2, e_2$)

    while $mid_1 < end_1$ and $mid_2 < e_2$ do
        if $l_1[mid_1] < l_2[mid_2]$ then
            $s_1 \leftarrow mid_1$
            $e_2 \leftarrow mid_2$
        else
            $e_1 \leftarrow mid_1$
            $s_2 \leftarrow mid_2$
        end if

        $mid_1 \leftarrow$ MID($s_1, e_1$)
        $mid_2 \leftarrow$ MID($s_2, e_2$)
    end while

    if $mid_1 \geq e_1$ then
        return $l_1[mid_1]$
    else
        return $l_2[mid_2]$
    end if
end function
```

Algorithm 3: Value of the n^{th} smallest integer in either list

Problem 3

Give an $O(n)$ average case running time algorithm that computes the following:

Kth-SMALLEST($list, k$)

Input: An unsorted list *list* of unique integers and an integer k
Output: The value of the k^{th} smallest integer from the list

Solution

The algorithm is based off of **QuickSort** and is often called **QuickSelect**[1]. The principle idea is to use the partitioning function of **QuickSort** because the element that is selected to be a partition will be placed in its correct place in the list. The position then will tell us where to look for the kth item.

```
function KTH-SMALLEST($list, k$)
    $index \leftarrow$ KTH-SMALLEST-REC($list, k, 0, list.length$)
    return $list[index]$
end function

function QUICKSELECT($list, k, start, end$)
    if $start \geq end$ then
        return $end$
    end if

    $partition \leftarrow$ PARTITION($list, k, start, end$)

    if $k < partition$ then
        return QUICKSELECT($list, start, partition - 1$)
    else if $k > partition$ then
        return QUICKSELECT($list, partition + 1, end$)
    end if

    return $list[k]$
end function
```

Algorithm 4: Give the k^{th} smallest integer from the list

Where the **Partition** algorithm is dependent on the type of partitioning scheme used.

Reference

1. Tim Roughgarden's lecture on Coursera.org: https://class.coursera.org/algo-004/lecture/36

Problem 4

Let T be a tree with n vertices. We say that a vertex v is a *minimal separator of* T if its removal splits T into two or more subtrees, each with at most $n/2$ vertices.

Part A
Show that every finite tree has at least one minimal separator.

Solution

Proof. We will use induction to show that every finite tree has at least one minimal separator.

Base Consider the trees with $n = 1$ and $n = 2$ vertices. These trees trivially have minimal separators because each vertex in the base cases can be a minimal separator.

Step We wish to show that for any tree composed of a finite number of vertices still has at least one minimal separator.

Now consider the trees T_i where k is some finite integer the number of subtrees and $T = \bigcup_{i=1}^{k} T_i$ and where $|T| = n + 1$.

By assuming the induction hypothesis, we can assume that every T_i has a minimal separator.

By waving our magic wand[1], we can clearly see that the total tree will always have a minimal separator. Thus this concludes our proof.

Happy happy, joy joy. This proof is hard. □

Reference

1. I could not figure out this proof, these people are smart: http://homes.cs.washington.edu/~jrl/papers/fhl-sicomp.pdf

Problem 4 continued on next page. . .

Part B
Show an $O(|V|)$ algorithm for identifying a minimal separator in a given tree.

Solution

```
function VISIT($T, u$)
    for all $v \in T.Adj[u]$ do
        if $v.state$ = unprocessed then
            $v.parent = u$
            VISIT($T, v$)
        end if
    end for
    $u.state$ = processed
end function

function MINIMALSEPARATOR($T$)
    for all $v \in T.V$ do
        $v.state$ = unprocessed
        $v.parent$ = NULL
    end for

    for all $v \in T.V$ do
        if $v.state$ = unprocessed then
            VISIT($T, v$)
        end if
    end for

    $x \leftarrow T.V[0]$
    $y \leftarrow x$
    while $x \neq$ NULL do
        $y \leftarrow x$
        $x \leftarrow x.parent$
    end while

    Recursively count children starting from the parent, when children of a given node equals $|T.V|$,
    return that node.                    // Counting children runs in $O(|V|)$ time
end function
```

Algorithm 5: Identify a minimal separator in the given tree

Problem 5

Give an algorithm that computes the following:

BST-Reconstruction(*traversal*)

Input: An array of elements generated by a pre-order traversal of some binary search tree.
Output: A binary search tree identical to the original tree.

Solution

The following algorithm works because a preorder listing of a binary tree is recursive in nature. The first element off the queue will be the root value. The left value will then be the next item on the queue, as long as it value is less than the current value.

Next we check to make sure the queue isn't empty, if it isn't, then we compare the value on the top of the queue to see if it is less than the parent. If the item isn't less than the parent, then it must belong in another subtree. If it does add continue the recursion.

The algorithm also assumes that the stack works such that when initialized with an array, the elements are popped off from the beginning and added to the beginning. Or in other words, the front of the array is the top of the stack.

```
function BST-Reconstruction($traversal$)
    $stack \leftarrow$ new Stack($traversal$)
    return BST-Reconstruction-Rec($stack, null$)
end function

function BST-Reconstruction-Rec($stack, parent$)
    $root \leftarrow$ new Node($stack.pop()$)

    if $stack.size() \neq 0$ then
        if $root.value \geq stack.peek()$ then
            $root.left \leftarrow$ BST-Reconstruction-Rec($stack, root.value$)
        end if
    end if

    if $stack.size() \neq 0$ then
        if $parent \geq stack.peek()$ or $parent == null$ then
            $root.right \leftarrow$ BST-Reconstruction-Rec($stack, parent$)
        end if
    end if

    return $root$
end function
```

Algorithm 6: Binary search tree reconstruction

Problem 6

Let $G = (V, E)$ be a connected, undirected graph.

Prove or disprove:

$$\exists v \in V \mid G' = (V \setminus \{v\}, E) \text{ is connected}$$

Solution

According to the definition of a connected graph, this means that for any two vertices, there exists a path from one to the other. A little more formally:

$$\forall u, v \in V \mid u \rightsquigarrow v$$

Note: The symbol $u \rightsquigarrow v$ means that there is a path from u to v.

To prove that G is still connected after removal of a vertex v, let's do a proof by contradiction.

Proof. We will use a proof by contradiction to prove that for a graph G, that is connected and undirected, there exists a vertex v such that when it is removed, it results in a graph G' that is still connected.

Assume for the sake of contradiction that when v is removed that this results in a graph that is no longer a single component and is not connected.

First we select vertices v and u such that $u \rightsquigarrow v$ is the longest path in the graph G.

Now if we were to remove v from the graph G, this would result in multiple components because of our original assumption. This means that there are two vertices, say s and t, that are separated when v is removed.

If the separation causes multiple components to form, then if we were to take the path $u \rightsquigarrow v \rightsquigarrow s$ or $u \rightsquigarrow v \rightsquigarrow t$, both of which cross the gap, this would create a longer path than $u \rightsquigarrow v$.

Thus since we assumed that $u \rightsquigarrow v$ is the longest path, this is a contradiction because it would no longer be the longer path.

Since we have found a contradiction, this means that if we select v such that it is at the end of the longest path of the graph G, we can safely remove it to ensure that G' is still connected. This concludes the proof. $\square$

Reference

1. Contradiction idea inspired by the proof in problem 4: http://dcg.epfl.ch/files/content/sites/dcg/files/Courses/Graph%20Theory%202011/problemset4sol_2011.pdf

Problem 7

A *mother* vertex in a directed graph $G = (V, E)$ is a vertex v such that all other vertices in G can be reached by a directed path from v.

Part A

Give an $O(n + m)$ algorithm to test whether a given vertex v is a mother of G, where $n = |V|$ and $m = |E|$.

Solution

The algorithm is as follows:

```
function IsMother($s, G$)
    BFS($G, s$)

    for all $v \in G.V$ do
        if Visited($v$) = False then
            return False
        end if
    end for

    return True
end function
```

Algorithm 7: Determine if a given vertex is a mother of a graph

Justification

BFS runs in $O(n + m)$ time and it visits all the nodes that have a path from the starting vertex. Thus when it is finished, we just need to see if there are any such vertices that haven't been visited yet.

Problem 7 continued on next page...

Part B

Give an $O(n + m)$ algorithm to test whether a graph G contains a mother vertex.

Solution

The algorithm is as follows:

```
function StronglyConnectedComponentGraph($G$)
    DFS($G$)                                   // DFS runs in $O(n + m)$ time
    $G^T \leftarrow$ Transpose($G$)            // $G$ but with the edges reversed, runs in $O(n + m)$ time
    DFS($G^T$)

    $G' \leftarrow$ new Graph( )
    for all $c \in$ Components($G$) do
        $n \leftarrow$ new Node( )
        AddNode($G', n$)                       // Represent each component as a single node
        AddOutEdges($G', c, n$)                // Add out going edges to the node $n$
    end for
    return $G'$
end function

function ContainsAMother($G$)
    $G' \leftarrow$ StronglyConnectedComponentGraph($G$)    // Runs in $O(n + m)$ time
    $sources \leftarrow 0$
    for all $v \in G'.V$ do                    // Iterate over all the vertices in the component graph
        if In-Degree($v$) $= 0$ then
            $sources \leftarrow sources + 1$
        end if
    end for
    if $sources = 1$ then                      // A single source vertex means it is a mother vertex
        return True
    else
        return False
    end if
end function
```

Algorithm 8: Determine if a given graph has a mother.

Reasoning The algorithm **StronglyConnectedComponentGraph** returns a graph such that all strongly-connected components are represented by a single vertex with edges only between each strongly-connected components. This means that the entire graph is directed and acyclic. Therefore to find a *mother vertex*, we need to check the new graph to see if it contains only a single source vertex. The reason a single source vertex in a dag means it is a *mother vertex* is because if more than one exist, than that means that the n source vertices can never reach each other and thus it isn't a *mother vertex*.

Reference

1. Algorithm idea inspired by Problem 5: http://www.cs.sunysb.edu/~skiena/373/hw/keys/key3.pdf
2. Algorithm **StronglyConnectedComponentGraph** inspired by *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein, pg. 617:

Problem 8

Suppose we are given the minimum spanning tree T of a given graph G and a new edge $e = (u, v)$ of weight w that we will add to G.

Give an $O(n)$ algorithm to find the minimum spanning tree of the graph $G + e$.

Solution

The algorithm is as follows:

```
function AddMST($T, e, G$)
    $u, v \leftarrow$ Vertices($e$)
    BFS($T, u$)
    $path \leftarrow$ Create-Path($u, v$)

    $largest \leftarrow path[0]$
    for all $edge \in path$ do
        if $w(edge) > w(largest)$ then
            $largest \leftarrow edge$
        end if
    end for

    if $w(e) < w(largest)$ then
        RemoveEdge(T, largest)
        AddEdge(T, e)
    end if
end function
```

Algorithm 9: Add an edge to a given minimum spanning tree

Note: The $w()$ function just determines the weight for the given edge.

Reasoning

The reason this works is because the algorithm **BFS** determines the path $u \rightsquigarrow v$ that is shortest. It then knows that adding the edge $e = (u, v)$ would create a cycle. Instead, it finds the largest edge in the existing path and removes that edge only if the given edge is smaller. This ensures that the MST at the end is still minimal given the edge.

Justification

The justification for why it runs in $O(n)$ time is because in a tree, the number of edges is always one less that the number of vertices. Therefore the run time of **BFS** is $O(n)$ and then we do one more possible loop over the path which could be $O(n)$ as well. This gives the final runtime of roughly $O(2n)$ which is of course still just $O(n)$.

Problem 9

Problem 6–7 from the text.

Part A

Let T be a minimum spanning tree of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Do the edges of T form a minimum spanning tree of G'? Prove the statement or give a counterexample.

Solution

Proof. We will prove that a minimal spanning tree T will still be a minimal spanning tree even after adding a weight of k to every edge in a graph.

To prove this, we will use the fact that for any tree with $|V|$ number of vertices, the tree will have $|V|-1$ edges.

Now given T is a minimal spanning tree, we know that the property of a minimal spanning tree tells us that the number of vertices, V is then equal to $V = |G.V|$. Given the above fact, we then know that the number of edges in T is then $E = |G.V| - 1$.

It then follows that for every such minimal spanning tree of the graph G, it is guaranteed to have E edges.

By adding k to every edge weight in G, overall we will be adding $|G.E| \cdot k$ total weight to the tree. But for every possible spanning tree in G, we will only be adding $E \cdot k$ weight to each spanning tree.

Since every single spanning tree will have E edges and then Ek weight added to it, it then follows by the definition of a minimal spanning tree that there cannot exist another spanning tree that is smaller than our current minimal spanning tree T because every single spanning tree has an increased total weight of Ek.

Thus since our minimal spanning tree is still minimal when adding k to every edge in G the proof is complete. □

Part B

Let $P = \{s, \ldots, t\}$ describe the shortest weighted path between vertices s and t of a weighted graph G. Construct a new graph G' by adding a weight of k to every edge of G. Does P describe a shortest path from s to t in G'? Prove the statement or give a counterexample.

Solution

Counterexample

P still does not necessarily describe a shortest path from s to t in G'. This is weight added to each path is dependent on how many sections are in the path. If we were to let $k = 5$ and add it to each path, a path that has 1 edge will have 5 added to it, but a path with 2 edges will get 10 added to it. This is illustrated in the below example.

The first figure, Figure 1 is G and that P is the two edges straight across the line of nodes, from s to t then to u. However the second graph is Figure 2 has had k added to each edge, the shortest path, P is now the single edge from s to u.

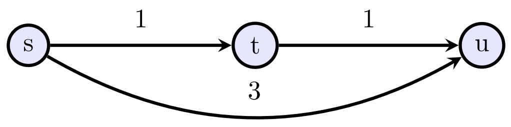

Figure 1: The graph G.

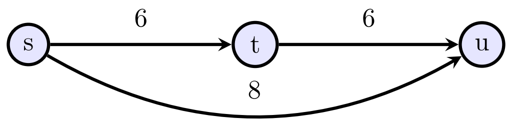

Figure 2: The graph G'.