Model Solutions
Assignment 2

November 9, 2014

1 Splitting a red-black tree in $O(\log n)$ time

As the question asks us to use $specialUnion(T_1, T_2)$, we take a closer look at that algorithm.

Understanding specialUnion

$T = specialUnion(T_1, T_2)$ merges two RBT, T_1 and T_2 (given $T_1 < T_2$) in $O(\max(h_1, h_2))$ time, where h_1 and h_2 are the black heights of T_1 and T_2, respectively. The bottleneck step here is finding minimum element x in T_2. Note that, if x is already known special union runs in $O(|h_1 - h_2|)$ time.
For this problem we modify $specialUnion(T_1, T_2)$ to
$T = specialUnionMod(T_1, x, T_2)$, where x is such that $T_1 < x < T_2$ and $T = T_1 \cup \{x\} \cup T_2$. In this modification, x is made the root of T and T_1 and T_2 are made x's left and right sub-tree respectively. This modification of $specialUnion$ runs in $O(|h_1 - h_2|)$ time.

Also, note that the height of T returned by $specialUnionMod$ is at most $\max(h1, h2) + 1 = O(\max(h1, h2))$.

1.1 Algorithm $split(T, x)$

$O(\log^2 n)$ algorithm

Using a naive approach we can achieve $O(\log^2 n)$ time. Initialize two empty trees T_1 and T_2. T_1 will store all the elements less than x and T_2 will store all the elements greater than x. Simply traverse from root to x and at every node, union the sub-tree not traversed with T_1 or T_2 appropriately using $specialUnion(T_1, T_2)$. There can be at most $O(\log n)$ union operations and each union operation will take at most $O(\log n)$ time.

$O(\log n)$ algorithm

We can improve the above naive strategy by using the modified specialUnion wisely. We note that using $specialUnionMod$ while traversing from root to

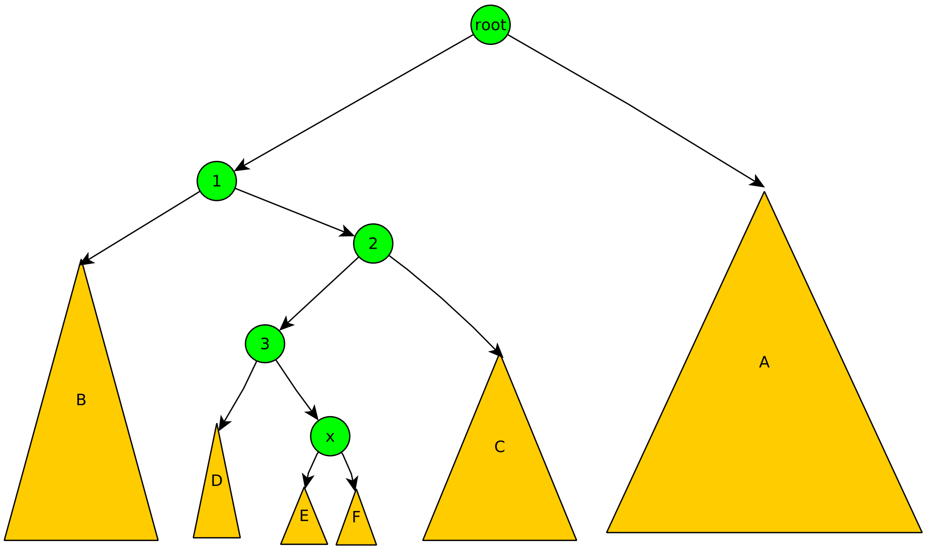

Figure 1: Iterationwise progression of split trees T_1 and T_2. Initialization: $T_1 = E$, $T_2 = F$, Iteration 1: $T_1 = E \cup D$, $T_2 = F$, Iteration 2: $T_1 = E \cup D$, $T_2 = F \cup C$, Iteration 3: $T_1 = E \cup D \cup B$, $T_2 = F \cup C$, Iteration 4: $T_1 = E \cup D \cup B$, $T_2 = F \cup C \cup A$

x will not give us much advantage because the height of T_1 and T_2 will be $O(\log n)$ on the first union itself and all the later union operations will take $O(\log n)$ time. The 'aha!' step is to note that a bottom-up traversal will build T_1 and T_2 incrementally, keeping the costs of union step in check. For example see figure 1

Pseudo Code

Algorithm 1 *split*(T, x)

//Search x in T

1: $currNode \leftarrow search(T, x)$

//Initialize T_1 and T_2

2: $T_1 \leftarrow \{\}$, $T_2 \leftarrow \{\}$

3: **if** $(currNode \neq null$ and $left(currNode) \neq null)$ **then**

4: $\quad T_1 = left(currNode)$

5: **end if**

6: **if** $(currNode \neq null$ and $right(currNode) \neq null)$ **then**

7: $\quad T_2 = right(currNode)$

8: **end if**

//Traverse from x to root and perform specialUnion on the way

9: **while** $parent(currNode) \neq currNode$ **do**

10: $\quad currNode \leftarrow parent(currNode)$

11: $\quad$ **if** $(value(currNode) < x)$ **then**

12: $\quad\quad T_1 \leftarrow specialUnionMod(T_1, currNode, left(currNode))$

13: $\quad$ **else**

14: $\quad\quad T_2 \leftarrow specialUnionMod(T_2, currNode, right(currNode))$

15: $\quad$ **end if**

16: **end while**

Analysis

1. **Sketch of correctness**: Initially T_1 contains left sub-tree of x and T_2 contain right subtree of x (line 4-7). We maintain following invariant: At the end of each iteration, we have correctly added the elements in sub-tree rooted at $currNode$ in the correct tree, T_1 or T_2.

   As we move from x to root, at any node ($currNode$ in pseudo code), all the elements in the sub-tree of $currNode$ containing x have already been added to either T_1 or T_2, in the previous iterations. In this iteration, if x is contained in the left sub-tree of $currNode$ we add $currNode$ and right sub-tree to T_2 as they are greater than x (line14). Otherwise, if x is contained in the left sub-tree of $currNode$ we add $currNode$ and left sub-tree to T_1 as they are smaller than x (line 12). We terminate when we reach the root node.

2. **Time complexity** Suppose for creating T_1, a sequence of unions of trees with following heights is performed h_1, $h_2, \ldots, h_k$, As we move from x to root we know $h_1 \leq h_2 \leq \ldots \leq h_k$.

   When we perform a union operation between trees of height a and b, the height of resultant tree is $O(\max(a, b))$ Therefore, height of T_1

follows the following sequence h_1, $h_2, \ldots, h_k$. Using this, we can easily claim that the i^{th} union operation takes $O(h_{i+1} - h_i)$ time. Hence time complexity for creating T_1 is

$$\sum_{i=1}^{O(\log n)} O(h_{i+1} - h_i) \;=\; O(\log n)$$

.

3. **Space Complexity** We just manipulate the pointers in the input tree. Therefore, only $O(1)$ extra space is required.

2 Thinking beyond limits and beyond marks

2.1 Finding $min\_L(s)$ for vertex s

We know a DFS from s will visit all the vertices reachable from s. We can then simply compare the label $L(v)$, $\forall v$ visited during call to DFS(s).

Pseudo Code

Algorithm 2 $DFS(G, s)$

for $v \in G.V$ **do**
    $visited[v] \leftarrow false$
end for
$min\_L(s) \leftarrow \infty$
$DFS\_visit(G, s, s)$

Algorithm 3 $DFS\_visit(G, v, source)$

if $L[v] < min\_L[source]$ **then**
    $min\_L(source) \leftarrow L[v]$
end if
$visited[v] \leftarrow true$
for $u \in G.V \; s.t. \; (v, u) \in G.E$ **do**
    if $visited[u] = false$ **then**
        $DFS\_visit(G, u, source)$
    end if
end for

Analysis

1. **Correctness**: Whenever a node is visited its label is compared with minimum label seen so far and value of min_L is updated. From correctness of DFS it follows that all nodes connected to s are visited.

2. **Time complexity**: Same as DFS $O(m+n)$
3. **Space complexity**: Same as DFS $O(n)$

2.2 Computing $min\_L(v)$ for all the vertex

The key idea is to observe that if a node v is reachable from a set of nodes W then the set W is reachable from v in the reverse graph G' (graph with same vertices as G, but with reversed edges). We note that $min\_label(v)$ of a node is the minimum label node reachable from v in G. Equivalent definition in the reverse graph G' for $min\_label(v)$ is the minimum label among the nodes that can reach v. Using this property if we start DFS from nodes in increasing order of label value we will be able to mark all the min labels.We follow the following procedure:

1. Create reverse graph G'
2. Sort nodes of G' in ascending order
3. Do DFS on G' but select source of DFS in the sorted order
4. Every node is visited exactly once and whenever a node v is visited mark $min\_L(v) \leftarrow L[source]$

Pseudo Code

Algorithm 4 $assgnMinLabel(G)$

//Reverse G
$G'.V \leftarrow G.V$
for $(u, v) \in G.E$ **do**
    $G'.E \leftarrow G'.E \cup \{(v, u)\}$
end for
$Sort(G'.V)$
for $v \in G'.V$ **do**
    $visited[v] \leftarrow false$
end for
//call $DFS\_visit(G', vertex, source)$
for $v \in G'.V$ **do**
    if $visited[v] = false$ **then**
        $DFS\_visit(G', s, s)$
    end if
end for

Algorithm 5 $DFS\_visit(G', v, source)$

$visited[v] \leftarrow true$
$min\_L[v] \leftarrow L[source]$
for $u \in G'.V\ s.t.\ (v, u) \in G'.E$ **do**
  if $visited[u] = false$ **then**
    $DFS\_visit(G', u, source)$
  end if
end for

Analysis

1. **Correctness**: In DFS every node is visited exactly once and once a node v is visited we update its $min\_L(v)$ as the label of the source s of the DFS call. As the source of the DFS call is selected in sorted order it implies that v is not reachable (in G') from any node with smaller label value than s. Also, if v is reachable from s in G' then s is reachable from v in G. Therefore, $min\_L(v)$ is the minimum possible label v can reach in the original graph G.

2. **Time Complexity**: Creating reverse graph takes $O(m + n)$ if adjacency list is used, sorting takes $O(n \log n)$ and DFS takes $O(m + n)$, therefore time complexity is $O(m + n \log n)$.

3. **Space Complexity**: $O(m + n)$ for storing reverse graph and $O(n)$ for DFS.

3 Breadth of special graph

We begin by pondering over the question of the longest path in a rooted tree. The first question that naturally arises is about the structure of such a path. Should the longest path always pass through the root node? It is instructive to look at figure 2 to get a better understanding. Here path P-N-L-J-H-F-I-K-M-O-Q is the longest path and it doesn't pass through the root A. For this example tree, $breadth[A]$ i.e. the length of the corresponding path in the sub-tree rooted at A is 10 and the longest path doesn't pass through A. From this we get the idea that the longest path for sub-tree rooted at A could either pass through A or it is present in one of the sub-trees rooted at its children B, C or D.

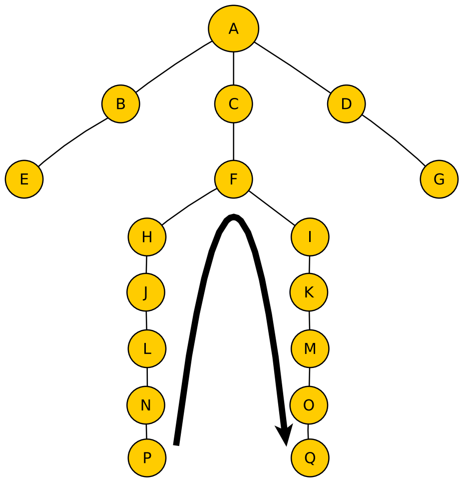

Figure 2: Example illustrating that the longest path doesn't necessarily pass through the root node

In general we should try to answer the question: for a node v, what is the relation between $breadth(v)$ and $breadth(u)$, $\forall u \in v.children$? Observe that it is the maximum of the longest paths in sub-tree rooted at each of its children and the longest path passing through v itself. Notice that the longest path through v is related to maximum length path from leaf nodes to v's children. Longest path through v is the sum of the top two maximum length paths (max_1, max_2) from root to two of v's children and 2. This gives rise to following recurrence:

$$breadth(v) = \max(breadth(u), max_1 + max_2 + 2), \forall u \in v.children$$

This recurrence becomes the crux of the recursive algorithm below. The last question that remains is that above discussion is for a rooted tree but input is given in form of a graph. This is answered by the fact that the given graph is a tree and a DFS from any vertex will give us a tree rooted at that vertex.

Algorithm 6 $DFS(G, s)$

for $v \in G.V$ **do**
  $visited[v] \leftarrow false$
end for
for some $s \in G.V$, do $DFS\_visit(G, s)$
Return max $breadth[v]$, $\forall v$ as breadth of G

Algorithm 7 $DFS\_visit(G, v)$

```
INPUT: Graph G and node to be visited v
OUTPUT:1) breadth of tree rooted at v
       2) max path from some leaf to v
max_1 ← 0, max_2 ← 0, breadth_v ← 0
visited[v] ← true
for u ∈ G.V s.t. (v, u) ∈ G.E do
  if visited[u] = false then
    (breadth_u, maxlen_u) = DFS_visit(G, u)
    breadth_v = max(breadth_u, breadth_v)
    if max_1 < maxlen_u + 1 then
      max_2 ← max_1
      max_1 ← maxlen_u + 1
    else if max_2 < maxlen_u + 1 then
      max_2 ← maxlen_u + 1
    end if
  end if
end for
breadth_v ← max(breadth_v, max_1 + max_2)
return (breadth_v, max_1)
```

Analysis

1. **Sketch of correctness:** Argue that the above recurrence is correct. Then, starting from the base case, inductively show that $DFS\_visit(G, v)$ computes correct $breadth_v$ for a sub-tree rooted at v and max path from leaf to v.

2. **Time Complexity**: Same as DFS, $O(m + n) = O(n)$ for tree graph

3. **Space Complexity**: Same as DFS, $O(n)$