Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dmytro Chibisov
Sandeep Sadanandan

Winter Semester 2007/08
Solution Sheet 9
January 08, 2008

Fundamental Algorithms

Problem 1

Prove that a binary tree can be reconstructed unambiguously using the numberings of a preorder and a postorder traversals of the tree. Can the same be done using preorder and inorder?

Solution

Assumption: In the sequence, if the there are nodes with just one child, then the non-existing child is represented with a symbol $\triangle$. Otherwise, we cannot differentiate the left/right child.

The preorder traversal gives the root, left-subtree and the right-subtree. Where as, the post order gives the left-subtree, right-subtree and root.

So, it is clear that the root of the tree is the first element in the preorder sequence (or the last number in the postorder sequence)

From the definition of traversals, it is evident that the root of the right subtree is the predecessor of the last element, in the postorder. Once we find out the occurance of that number in the preorder sequence, all the numbers after the specified number will be on the right subtree and the numbers before it will be on the left subtree.

Similarly, the number coming next to the root node in the preorder is the root of the left subtree. All the numbers in the postorder sequence till the occurance of this specified number are in the left subtree. And the numbers from then are in the right subtree.

We have already found out the root and we now have the preorder and postorder traversals of the left and right subtrees. Using the same method, we can find out the roots and subtrees recursively.

The details are marked in the figure below. (please refer to the example given in the class)

If preorder and inorder traversals are given, the method is more easy. The first number in preorder sequence is the root. On finding out that number in the inorder sequence, all the elements to the left of it in the inorder sequence are in the left tree and the ones on right are in the right subtree.

From this knowledge, we can mark the preorder traversals of left and right subtrees in the given preorder traversal of the tree. Once we have preorder and inorder traversals of

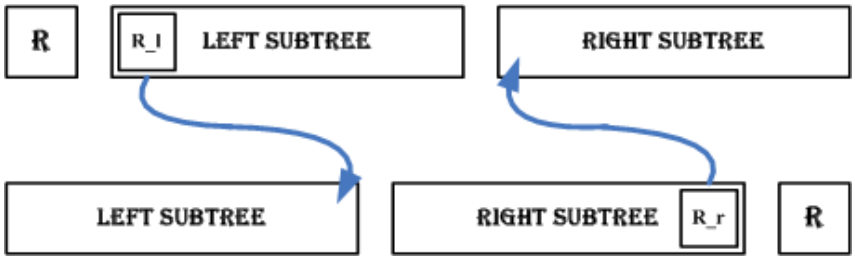

Abbildung 1: Locating the root and left/right subtrees from preorder and postorder sequences

the left and right subtrees, the method could be applied recursively. The following figure gives some details.

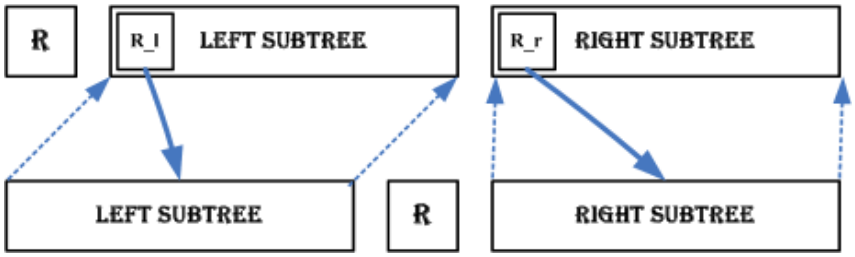

Abbildung 2: Locating the root and left/right subtrees from preorder and inorder sequences

Problem 2

For the given graph, starting with node 0, show how BFS and DFS traversals are done.

Solution

The following piture shows the call tree created by BFS. The nodes shown in same color are at the same level. The edges going from one level are marked in same color too.
The table below shows the contents of the queue while search was done.

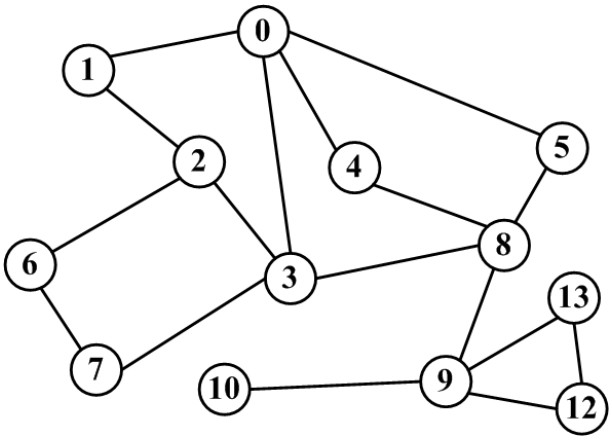

Abbildung 3: The Graph for DFS/BFS

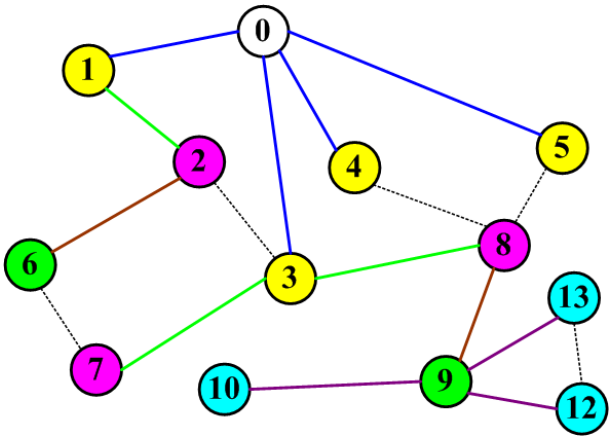

Abbildung 4: BFS on the Graph

Step	Visited/Done	In the queue
1		0
2	0	1, 3, 4, 5
3	0, 1	3, 4, 5, 2
4	0, 1, 3	4, 5, 2, 7, 8
5	0, 1, 3, 4	5, 2, 7, 8
6	0, 1, 3, 4, 5	2, 7, 8
7	0, 1, 3, 4, 5, 2	7, 8, 6
8	0, 1, 3, 4, 5, 2, 7	8, 6
9	0, 1, 3, 4, 5, 2, 7, 8	6, 9
10	0, 1, 3, 4, 5, 2, 7, 8, 6	9
11	0, 1, 3, 4, 5, 2, 7, 8, 6, 9	10, 12, 13
12	0, 1, 3, 4, 5, 2, 7, 8, 6, 9, 10	12, 13
13	0, 1, 3, 4, 5, 2, 7, 8, 6, 9, 10, 12	13
14	0, 1, 3, 4, 5, 2, 7, 8, 6, 9, 10, 12, 13	

The following figure shows the DFS-tree. Here the graph starts with node 0 and the sequence of nodes are marked near the node.

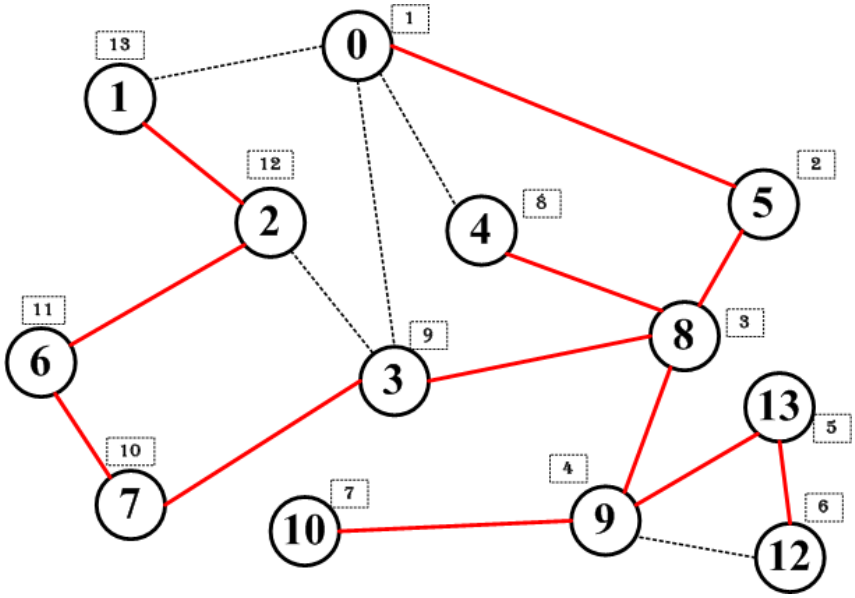

Abbildung 5: DFS on the Graph

Problem 3

Show that the tree defined by the edges traversed in a BFS (starting at v_0) is a shortest paths tree rooted at v_0.

Solution

A complete mathematical proof based on induction is available on many texts and also available online. But that appears to be out of the scope of our course. The following proof give a more verbal treatment.

BFS lists all the vertices at level $k-1$ before those at level k. Therefore, it will place into the queue all vertices at level k before all those of level $k+1$ and therefore list the ones at k before those in level $k+1$. It is not possible for two vertices which are connected and have a difference of levels to be more than 1. ie, if a node is at level i and a connected node cannot be in level $i+2$. Because if they are connected, then that node should be added at level $i+1$.

So BFS actually gives a shortest path tree starting at root.

- Every vertex has a path from/to root.
- The path length is equal to the level
- No path can skip a level hence the level will be always the minimum possible.

Hence the available path will be minimum path - hence the shortest paths tree.

Problem 4

Design an algorithm to find out the k^{th} smallest number from a set of n unsorted (pair-wise different) numbers. What is the complexity of the algorithm?

Solution

This problem has more than one solution. Let's start from the most naive method possible.

1. Sweep through the numbers k times to have the desired element.

   This method is the one used in bubble sort, every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed k times. So the complexity is $O(n * k)$.

2. Sort and take k^{th} element.

   Step 1: Sort the numbers

   Step 2: Pick the k^{th} element

   The complexity is very evident. Sorting of n numbers is of $O(n \lg n)$. And picking k^{th} element is of $O(k)$. So the total complexity is $O(n \lg n + k) = O(n \lg n)$.

3. Use a tree to sort

   Step 1: Insert the elements to a binary search tree

   Step 2: Do an inorder traversal until and print k elements which will be the smallest ones. So, we have the k^{th} smallest element.

   The cost of creation od a binary search tree of n elements is $O(n \lg n)$. And the traversal upto k elements is $O(k)$. Hence the complexity is $O(n \lg n + k) = O(n \lg n)$.

   Pitfall: If the numbers are sorted in descending order, we will be getting a tree which will be skewed towards left. In that case, construction of the tree will be $0 + 1 + 2 + \ldots + (n-1) = \frac{(n-1)*n}{2}$ which is $O(n^2)$.

   To escape from this pitfall, we can keep the tree balanced, so that the cost of constructing the tree will be only $n \lg n$.

4. Use a smaller tree to give the same result.

   Step 1: Take the first k elements of the sequence to create a balanced tree of k nodes. (this will cost $k \lg k$)

   Step 2: Take the remaining numbers one by one, and

   - if the number is larger than the largest element of the tree, DO NOTHING
   - if the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is $\lg k$ since the tree is a balanced tree of k elements.

   Once the step 2 is over, the balanced tree with k elements will be having the smallest k elements. The only remaining task is to print out the largest element of the tree.

   Complexity:

   (a) For the first k elements, we make the tree. Hence the cost is $k \lg k$

(b) For the rest $n - k$ elements, the complexity is of $O(\lg k)$. That is step 2 has a complexity of $(n - k) \lg k$.

The total cost is $k \lg k + (n - k) \lg k = n \lg k$ which is $O(n \lg k)$. This bound is actually better than the ones provided earlier.

5. Use a random element to partition the set of numbers. (*You could skip this one*)

   This method is very simple, but the worst case complexity could go to $O(n^2)$. But on an average case, we get a linear complexity.

   Step 1: Take a random element from the set S of numbers

   Step 2: Partition the set S to three sets S_1, S_2 and S_3 such that each contain numbers less-than, equal-to, larger-than the random-element respectively.

   Step 3: If S_1 has more than k elements, then the k^{th} smallest element must be in S_1, apply the procedure recursively.

   Step 4: If S_1 has lesser than k elements, but S_1 and S_2 together has more than k elements, then the random element is the seeked-for k^{th} smallest element.

   Step 5: Id S_1 and S_2 together has more than k elements, then the element has to be there in S_3. Then search for $(k - |S_1| - |S_2|)^{th}$ element in S_3. (Make sure you understand why we reduced the value of k).

   This method actually partitions S to small fractions and the problem is broken down into a smaller problem every time. On an average case, the complexity is linear.