Longest path in an undirected tree with only one traversal

There is this standard algorithm for finding longest path in undirected trees using two depth-first searches:

- Start DFS from a random vertex v and find the farthest vertex from it; say it is v'.
- Now start a DFS from v' to find the vertex farthest from it. This path is the longest path in the graph.

The question is, can this be done more efficiently? Can we do it with a single DFS or BFS?

(This can be equivalently described as the problem of computing the diameter of an undirected tree.)

algorithms graphs trees

edited Jan 19 '16 at 10:43

Raphael ♦
55.4k 21 135 296

asked Apr 12 '13 at 14:00

emmy
634 1 6 17

2 What you are after is also called the *diameter* of the tree. (On trees, "longest shortest path" and "longest path" are the same thing since there's only one path connecting any two nodes.) – Raphael ♦ Apr 12 '13 at 15:06

3 Answers

We perform a depth-first search in post order and aggregate results on the way, that is we solve the problem recursively.

For every node v with children $u_1, \dots, u_k$ (in the search tree) there are two cases:

- The longest path in T_v lies in one of the subtrees $T_{u_1}, \dots, T_{u_k}$.
- The longest path in T_v contains v.

In the second case, we have to combine the one or two longest paths from v into one of the subtrees; these are certainly those to the deepest leaves. The length of the path is then $H_{(k)} + H_{(k-1)} + 2$ if $k > 1$, or $H_{(k)} + 1$ if $k = 1$, with $H = \{h(T_{u_i}) \mid i = 1, \dots, k\}$ the multi set of subtree heights[1].

In pseudo code, the algorithm looks like this:

```
procedure longestPathLength(T : Tree) = helper(T)[2]

/* Recursive helper function that returns (h,p)
 * where h is the height of T and p the length
 * of the longest path of T (its diameter) */
procedure helper(T : Tree) : (int, int) = {
  if ( T.children.isEmpty ) {
    return (0,0)
  }
  else {
    // Calculate heights and longest path lengths of children
    recursive = T.children.map { c => helper(c) }
    heights = recursive.map { p => p[1] }
    paths = recursive.map { p => p[2] }

    // Find the two largest subtree heights
    height1 = heights.max
    if (heights.length == 1) {
      height2 = -1
    } else {
      height2 = (heights.remove(height1)).max
    }

    // Determine length of longest path (see above)
    longest = max(paths.max, height1 + height2 + 2)

    return (height1 + 1, longest)
  }
}
```

1. $A_{(k)}$ is the k-smallest value in A (order statistic).

edited Mar 29 '15 at 21:28

ke.
5 2

answered Apr 12 '13 at 15:05

Raphael ♦
55.4k 21 135 296

@JeffE Regarding the second comment: Indeed, and this is taken care of in the last row: `height1 + height2` is the length of this path. If it is indeed the longest path, it is chosen by `max`. It's also explained in the text above, so I don't quite see your problem? Surely you have to recurse in order to find out whether it is indeed the longest path, and even if not it does not hurt (w.r.t. correctness) to recurse. – Raphael ♦ Apr 24 '13 at 10:30

@JeffE Regarding the first comment, the calculation for `height2` explicitly removes `height1` from

@JeffE Regarding the first comment, the calculation for `height2` explicitly removes `height1` from consideration, so how can it choose the same child twice? That, also, has been explained in the introductory text. – Raphael ♦ Apr 24 '13 at 10:44

1 Apparently, we speak different pseudocode dialects, because I have a hard time understanding yours. It would help to add an explicit English declaration that `longestPathHeight(T)` returns a pair `(h,d)`, where `h` is the height of `T` and `d` is the diameter of `T`. (Right?) – JeffE Apr 24 '13 at 15:14

@JeffE Right; I thought that was clear from the code, given the explanation, but apparently my extrapolation of "clear" for other pseudocode-paradigms was insufficient (mine is Scalaesque, I guess). Sorry for the confusion, I'm clarifying the code (hopefully). – Raphael ♦ Apr 24 '13 at 17:35

This can be solved in a better way. Also, we can reduce the time complexity to O(n) with a slight modification in the data structure and using an iterative approach. For a detailed analysis and multiple ways of solving this problem with various data structures.

Here's a summary of what I want to explain in a blog post of mine:

Recursive Approach – Tree Diameter Another way of approaching this problem is as follows. As we mentioned above that the diameter can

1. completely lie in the left sub tree or
2. completely lie in the right sub tree or
3. may span across the root

Which means that the diameter can be ideally derived by

1. the diameter of left tree or
2. the diameter of right tree or
3. the height of left sub tree + the height of right sub tree + 1 (1 to add the root node when the diameter spans across the root node)

And we know that the diameter is the lengthiest path, so we take the maximum of 1 and 2 in case it lies in either of the side or wee take 3 if it spans through the root.

Iterative Approach – Tree Diameter

We have a tree, we need a meta information with each of the node so that each node knows following:

1. The height of its left child,
2. The height of its right child and
3. The farthest distance between its leaf nodes.

Once each node has this information, we need a temporary variable to keep track of the maximum path. By the time the algorithm finishes, we have the value of diameter in the temp variable.

Now, we need to solve this problem in a bottom up approach, because we have no idea about the three values for the root. But we know these values for the leaves.

Steps to solve

1. Initialize all the leaves with leftHeight and rightHeight as 1.
2. Initialize all the leaves with maxDistance as 0, we make it a point that if either of the leftHeight or rightHeight is 1 we make the maxDistance = 0
3. Move upward one at a time and calculate the values for the immediate parent. It would be easy because now we know these values for the children.
4. At a given node,
   - assign leftHeight as maximum of (leftHeight or rightHeight of its left child).
   - assign the rightHeight as maximum of (leftHeight or rightHeight of its right child).
   - if any of these values (leftHeight or rightHeight) is 1 make the maxDistance as zero.
   - if both the values are greater than zero make the maxDistance as leftHeight + rightHeight – 1
5. Maintain the maxDistance in a temp variable and if in step 4 the maxDistance is more than the current value of the variable, replace it with the new maxDistance value.
6. At the end of the algorithm the value in maxDistance is the diameter.

edited Mar 20 '15 at 13:09

Raphael ♦
55.4k 21 135 296

answered Mar 18 '15 at 19:20

dharam
170 1 4

1 What does this add over my older answer, besides being less general (you only deal with binary trees)? – Raphael ♦ Mar 20 '15 at 13:10

7 This answer is more readable and easier to understand in my opinion (your pseudocode is very confusing). – reggaeguitar Mar 23 '15 at 22:49

Below is code that returns a diameter path using only a single DFS traversal. It requires extra space to keep track of the best diameter seen so far as well as the longest path beginning at a particular node in the tree. This is a dynamic programming approach based on the fact that a longest diameter path either doesn't include root, or is combination of the two longest paths of root's neighbours. Thus we need two vectors to keep track of this information.

```
 int getDiam(int root, vector<vector<int>>& adj_list, int& height, vector<int>&
path, vector<int>& diam) {
    visited[root] = true;
    int m1 = -1;
    int m2 = -1;
    int max_diam = -1;
    vector<int> best1 = vector<int>();
    vector<int> best2 = vector<int>();
    vector<int> diam_path = vector<int>();
    for(auto n : adj_list[root]) {
        if(!visited[n]) {
            visited[n] = true;
            int _height = 0;
            vector<int> path1;
            vector<int> path2;
            int _diam = getDiam(n, adj_list, _height, path1, path2);
            if(_diam > max_diam) {
                max_diam = _diam;
                diam_path = path2;
            }
            if(_height > m1) {
                m2 = m1;
                m1 = _height;
                best2 = best1;
                best1 = path1;
            }
            else if(_height > m2) {
                m2 = _height;
                best2 = path1;
            }
        }
    }

    height = m1 + 1;

    path.insert( path.end(), best1.begin(), best1.end() );
    path.push_back(root);

    if(m1 + m2 + 2 > max_diam) {
        diam = path;
        std::reverse(best2.begin(), best2.end());
        diam.insert( diam.end(), best2.begin(), best2.end() );
    }
    else{
        diam = diam_path;
    }


    return max(m1 + m2 + 2, max_diam);
}
```

answered May 1 '17 at 15:03

Trevor Van Loon

1 ♂ 1

2 This is not a coding site. We discourage answers that consist of primarily a block of code. Instead, we want answers that explain the ideas behind the algorithm, and give concise pseudocode (that don't require knowledge of any particular programming language to understand). How do you compute the longest path beginning at a particular node in the tree? (especially since the longest path might begin by going "up" the DFS tree, i.e., back towards the root) – D.W. ♦ May 1 '17 at 15:39