Technical Report CS74032-R

On the Minimal Total Path Length of a Spanning Tree

Andy N. C. Kang

David A. Ault

Department of Computer Science

College of Arts and Sciences

Virginia Polytechnic Institute & State University

Blacksburg, Virginia 24061

December 1974

Abstract. The notions of a balance node and the total path length with respect to a node u of a spanning tree are defined. We show that the total path length of a spanning tree with respect to u is minimal if and only if u is a balance node. An algorithm is also given to locate a balance node. A proof of the correctness of the algorithm is given and the complexity of the algorithm is analyzed.

Key words. spanning tree, total path length, balance node, proof of correctness and analysis of complexity of an algorithm

1. Introduction

Let T be a spanning tree of a graph and let u be a node in T. The total path length of T with respect to u is defined as the sum of the path lengths from each node in T to u. With respect to different nodes the total path length of T is usually different. We are interested in locating a node such that the total path length with respect to that node is a minimum among all total path lengths with respect to the nodes of T. In this paper, the notion of a balance node of T is defined and a characterization theorem is given which states that the total path length of T with respect to a node u is minimum if and only if u is a balance node of T. A linear algorithm is also given to locate a balance node. A proof of correctness of the algorithm is presented and the algorithm's complexity is analyzed.

2. Definitions and Preliminary Observations

Let T be a spanning tree of a graph. The nodes in T are denoted by small letters (possibly with a subscript) such as u, v_i, etc., the edges are denoted by node pairs such as (u, v_i), and the distance associated with the edge (u, v_i) is denoted by $d_{(u, v_i)}$.

Definition 1. The path length $\ell_{(u, v)}$ from node u to node v is defined as follows:

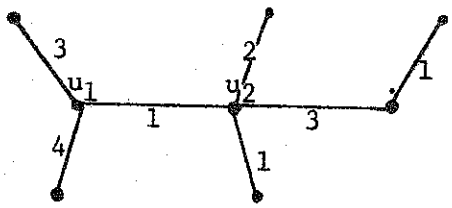
$$\ell_{(u,\,v)} = \begin{cases} 0 & \text{if } u = v \\ d(u,\,v) & \text{if } (u,\,v) \text{ is an edge of } T \\ \sum_{i=1}^{n} d(x_{i-1},\,x_i) & \text{if there exist nodes } x_0, x_1, \ldots, x_n \text{ such that } u = x_0,\ v = x_n \text{ and for all } i,\ 1 \le i \le n,\ (x_{i-1},\,x_i) \text{ are edges in } T. \end{cases}$$

Definition 2. The total path length of T with respect to u is defined as the sum of the path lengths from each node in T to u.

Definition 3. The minimal total path length of T is

$$\min_{u \in T} \{\text{total path length of T with respect to u}\}.$$

Example 1. Consider the following spanning tree.



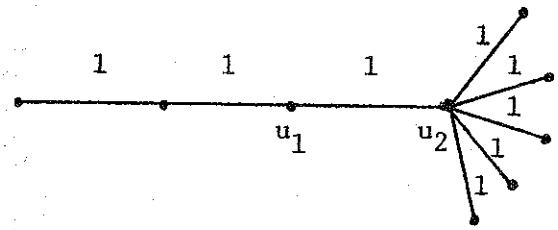
The total path length with respect to u_1 is $3 + 4 + 1 + (1 + 2) + (1 + 3) + (1 + 1) + (1 + 3 + 1) = 22$ and the total path length with respect to u_2 is $(3 + 1) + (4 + 1) + 1 + 2 + 3 + 1 + (3 + 1) = 20$. By an exhaustive search, we see that the minimal total path length is also 20.

It is quite natural to believe the center* [3] of a spanning tree to be the node such that the total path length with respect to it is minimal. To see that this is not true in general, consider the following spanning tree.

---

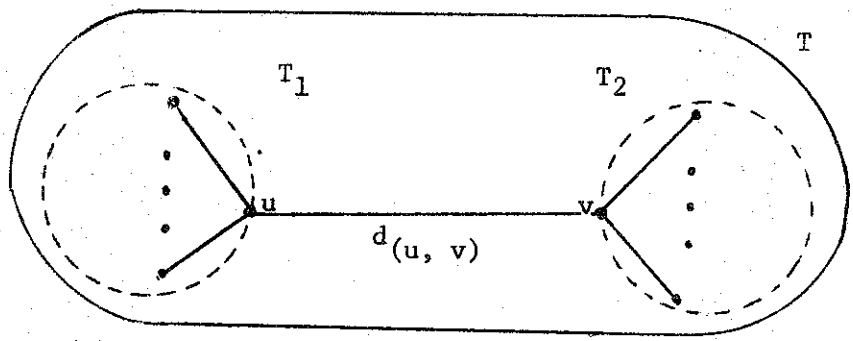
* A node c is a center of a tree T if $\max_{v \in T} \{\ell_{(c,\,v)}\} = \min_{u \in T} \{\max_{v \in T} \{\ell_{(u,\,v)}\}\}$.
Intuitively, c is the moddle node of T.

Example 2.



The center is u_1 and the total path length with respect to it is 14, while with respect to node u_2, the total path length is 11. Thus, u_2 is obviously a better choice for our purpose.

Let us consider the following spanning tree T.



Let (u, v) be an edge in T. If we delete (u, v) from T, we have two subtrees T_1 (which contains u) and T_2 (which contains v). Let n_1 (n_2) be the number of nodes in T_1 (T_2) and L_1 (L_2) be the total path length of T_1 (T_2) with respect to u (v), then the total path length of T with respect to u is $L_1 + L_2 + d_{(u, v)} \cdot n_2$. In this formula, L_1 is the path length contributed by the nodes in T_1 and $L_2 + d_{(u, v)} \cdot n_2$ is the path length contributed by the nodes in T_2, since the path length with respect to u from a node in T_2 is $d_{(u, v)}$ plus the path length of the same node with respect to v and there are n_2 nodes in T_2. Therefore, the total path length with respect to u is $L_1 + (L_2 + d_{(u, v)} \cdot n_2)$.

Similarly, the total path length with respect to v is $L_1 + d_{(u, v)} \cdot n_1 + L_2$. Thus, we have that $n_1 \geq n_2$ if and only if the total path length $(L_1 + L_2 + d_{(u, v)} \cdot n_2)$ with respect to u is less than or equal to the total path length $(L_1 + d_{(u, v)} \cdot n_1 + L_2)$ with respect to v. Let us state this fact in the following lemma.

Lemma 1. Let (u, v) be an edge of a spanning tree T. Let T_1 (T_2) be the subtree containing u (v) obtained by deleting (u, v) from T and let the number of nodes in T_1 (T_2) be n_1 (n_2), then $n_1 \geq n_2$ if and only if the total path length of T with respect to u is less than or equal to the total path length of T with respect to v.

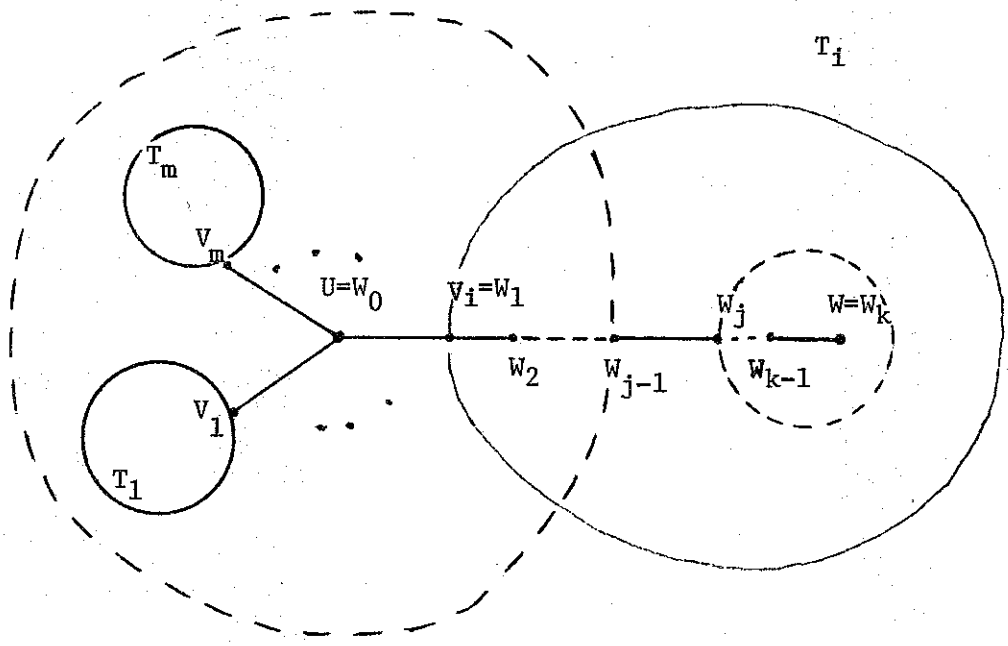
3. Balance Node of a Spanning Tree

Before we present the main result, we have the following definition.

Definition 4. Assume T is a spanning tree with n nodes. Let $v_1, v_2, \ldots, v_m$ be the nodes adjacent to a node u in T. For each i, $1 \leq i \leq m$, let T_i (T_i contains v_i) be the subtree obtained by deleting the edge (u, v_i) from T and let n_i be the number of nodes in T_i. We say that node u is a balance node of T if $n_i \leq n - n_i$ (equivalently $n_i \leq n/2$) for all i, $1 \leq i \leq m$.

Theorem 1. The node u is a balance node of a spanning tree T if and only if the total path length with respect to u is the minimal total path length of T.

Proof. Let us consider the following picture.



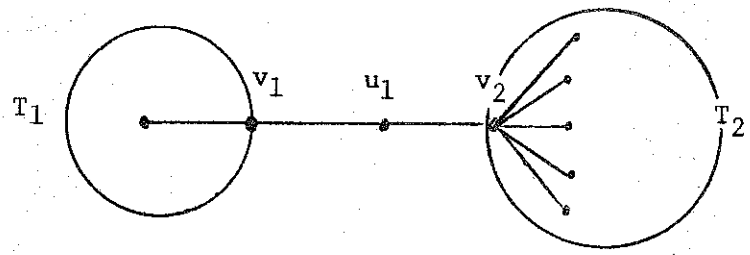
Let $v_1, v_2, \ldots, v_m$ be the adjacent nodes of u and for each i, $1 \leq i \leq m$, let T_i (containing v_i) be the subtree obtained by deleting the edge (u, v_i) from T.

Assume u is a balance node of T. Let us take an arbitrary node $w \neq u$ (say w is in T_i). Since T is a spanning tree, there exists a path from u to w. Assume the path is $(w_0, w_1, \ldots, w_k)$, i. e., $w_0 = u$, $w_1 = v_i$, $w_k = w$, and (w_{j-1}, w_j), $1 \leq j \leq k$, are edges in T. Let us consider the subtree of T_i containing w_j (the subtree is within the dotted line in T_i). The number of nodes in this subtree (denoted by s_j) is less than or equal to n_i which is the number of nodes in T_i. By the assumption that u is a balance node, we know that $n - n_i \geq n_i$. It follows that $n - s_j \geq s_j$. As a consequence of Lemma 1, the total path length with respect to w_{j-1} is less than or equal to the total path length with respect to w_j. Since this is true for any j, $1 \leq j \leq k$, it follows that the total path length with respect to u $(=w_0)$ is less than or equal to the total path length with respect to w $(=w_k)$. This shows that the total path length with respect to u is minimal.

Assume that the total path length with respect to u is minimal. Let

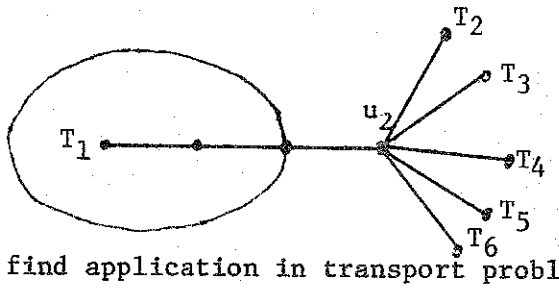
v_i be an adjacent node of u. If the total path length with respect to u is minimal, then the total path length with respect to u is less than or equal to the total path length with respect to v_i. As a result of Lemma 1, the number of nodes in T_i is less than or equal to the number of nodes not in T_i, namely, $n_i \leq n - n_i$. This is true for all i, $1 \leq i \leq m$. By definition, u is a balance node of T. Q.E.D.

Example 3a. Consider the spanning tree.



The number of nodes in T is 9 and the number of nodes in T_2 is 6 which is greater than 9 - 6 = 3. So u_1 is not a balance node and the total path length with respect to u_1 is not minimal.

Example 3b. In the following spanning tree, u_2 is clearly a balance node. The total path length with respect to u_2 is minimal.



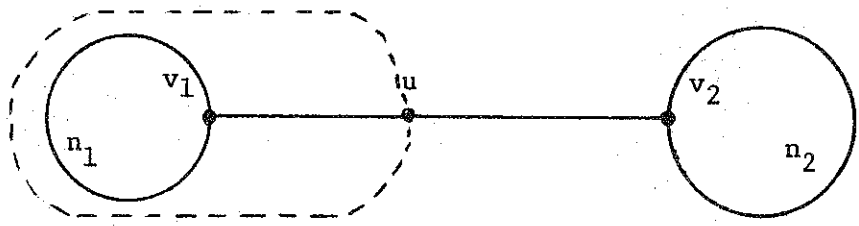
These concepts find application in transport problems. The situation to which they apply involves the movement from a central station to any one of a number of stations along predefined routes. The key restriction is that the movement to a given station, while possibly passing through other stations, must have as its only goal the arrival at the chosen

station. For example, consider the delivery of heavy machinery from a central plant or store yard to construction sites. Assume only one piece of machinery can be delivered per truck and that there is no priority for delivery to any station, then locating the store yard at the balance node will minimize the total miles traveled to make the deliveries.

Before we present an algorithm which locates a balance node, we have the following observation.

Theorem 2. In a spanning tree T, there are at most two balance nodes and they are adjacent.

Proof. If there are two balance nodes v_1 and v_2 which are not adjacent, then the path which connects them must contain at least three nodes, say, v_1, u, and v_2.



Let n_1 (n_2) denote the number of nodes in the subtree which contains v_1 (v_2). Since v_2 is a balance node, it follows from Theorem 1 that the number of nodes in the subtree containing u (the outer dotted line) is less than or equal to n_2. Further, n_1 is less than (by at least 1) the number of nodes in the subtree containing u. As a result, $n_1 < n_2$. Similarly, $n_2 < n_1$. This is a contradiction.

We assume that there are more than two balance nodes in T. Then at least two of these nodes cannot be adjacent to each other since all of

the nodes are connected by a spanning tree, which is an acyclic graph. Q.E.D.

4. Locating a Balance Node of T

We give a procedure which will determine in linear time the balance node of a given spanning tree and we verify that the algorithms used in the procedure are correct.

We assume that the following functions are provided to us to handle the manipulation of the data structure which represents the spanning tree. Several representations of graphs and techniques for their manipulation can be found in [1]. Some nodes of the spanning tree will be visited during the execution of the algorithm. Let SUBNODEOF be a function of one variable which returns a previously unvisited node which is adjacent to the node given as an argument. We assume that the node returned is chosen according to the Depth-First Search method as described in [4]. This method selects nodes at lower and lower levels in the tree until a leaf node is reached. It then selects a node by returning up the tree until the first new path down the tree can be found. Let PARENTOF be a function of one variable which returns the parent node of the node given as an argument.

We first describe a procedure BALANCE which determines a balance node of a spanning tree. BALANCE has one argument u which may be any node in the spanning tree. Upon return, u will be a balance node of the spanning tree. This procedure calls a function procedure LABEL which has as an argument a root node of a subtree and among other things it returns the

number of nodes in that tree. The details of the procedure LABEL will be given later. The procedure BALANCE uses a variable "tested" to record the number of nodes that have been considered in the search for a balance node. This definition implies that if the number of nodes tested is greater than or equal to $n/2$, then the current node u is a balance node because no untested node can be the root node of a tree of more than $n/2$ nodes.

```
procedure BALANCE (u);
begin
     tested ← 1;
     while tested < n/2 do
          begin
               v ← SUBNODEOF(u);
               nodecount ← LABEL (v);
               if nodecount > n/2 then
                    begin
                         u ← v;
                         tested ← n - nodecount + 1
                    end
               else tested ← tested + nodecount
          end
end
```

The algorithm seeks out a balance node by choosing an adjacent node v to the given node u and by using the function LABEL to determine the number of nodes in the subtree with root node v. If this tree contains

more than $n/2$ nodes, then u is the root node of a subtree with n - nodecount < $n/2$ nodes. Therefore, let v be a new candidate for a balance node. In this case we have tested (either directly or by elimination) (n - nodecount) + 1 nodes. If there are no more than $n/2$ nodes in the subtree, then count these with the tested nodes as part of a subtree of the yet to be determined balance node. Continue testing subnodes of the candidate for a balance node until at least $n/2$ nodes have been tested. The current choice of u is a balance node of the spanning tree.

We use the Invariant Relation theorem [2, p. 198] to show that the algorithm is correct. We restate the theorem here for the convenience of the reader. The purpose of the theorem is to verify the correctness of the computation in a while loop of the form

while C do S

where C is a condition and S is a statement (or a sequence of statements as in a begin end block). Suppose that P & Q are relations about the variables used in the loop, then the notation

P | S | Q

means: If P is true before execution of the statements S, then Q is true after execution of S. The theorem can now be stated.

Invariant Relation Theorem. Provided the loop halts, R&C | S | R implies

R | while C do S | R&¬C

R is called an invariant relation of the loop.

This theorem may be used to show the correctness of a while loop in a given procedure by finding a relation R among the variables in the loop such that R describes the initial state before entering the loop, R is true after execution of the statements of the loop and R&¬C imply

the desired results of the loop.

Theorem 3. The procedure BALANCE locates a balance node of a spanning tree.

Proof. An invariant relation R of the while loop is the following:

Given a node u of the spanning tree T and the nodes v_i adjacent to u, each v_i falls into one of two classes.

class 1. The subtree with root node v_i is known to have no more than $n/2$ nodes.

class 2. The node v_i has not been tested.

The condition C is the following: tested $\geq n/2$. Before the while loop begins, all of the v_i belong to class 2. After the loop halts, the condition R&$\neg$C implies that all of the v_i which are not in class 1 are untested nodes. Since tested $\geq n/2$, these nodes can only be root nodes of subtrees of less than or equal to $n/2$ nodes, hence they actually belong to class 1. By definition, node u is a balance node.

It remains to be shown that R is an invariant relation of the loop and that the loop halts. Assume that R holds before entering the begin end block of the while loop. A subnode v of u is selected. Either the number of nodes in the subtree with root node v is less than or equal to $n/2$, in which case v is one of the v_i in class 1, or there are more than $n/2$ nodes. In the latter case the node v becomes the candidate node u' and the previous node u is one of the v_i in class 1 with respect to u', since u can be the root node of no more than $n/2$ nodes. By relabeling the node u' with the label u, the relation R holds at the end of the begin end block. The while loop halts because tested is increased by at least

1 in every loop and $n/2$ is a fixed quantity throughout the procedure. Q.E.D.

The principle function of the procedure LABEL is to return the number of nodes in a given tree. Because the number of nodes in the subtrees of the given tree may be requested by the procedure BALANCE, an additional cell is associated with each node of the original spanning tree. In the process of determining the number of nodes in the tree, the number of nodes in each subtree of that tree is determined (at no extra cost) and this number is saved in the cell associated with the root node of that subtree. If LABEL is called with one of these root nodes as an argument, the number of nodes in that subtree is already known and can be returned with no further processing needed. The variables used in the procedure are

count — an array subscripted by the name of each node. It is used to record the number of nodes in the subtree formed by considering that node as a root node. It is initially set to zero.

counter — a variable used to accumulate the number of nodes in the subtree below each node.

level — a variable used to denote the depth to which the algorithm is operating in the subtree.

Let SUBNODE be a boolean function which returns true when there exists a previously unvisited node attached to the current node and false otherwise. The following definitions are made to aid in the discussion of the procedure LABEL.

Definition 5. A visited subtree V of a tree T is a subtree of T such that

the cell count(node) is positive for every node in the subtree.

Definition 6. A tree is labeled if, for each node v of the tree, count(v) is equal to the number of nodes in the subtree with root node v.

When LABEL is called with a node of the spanning tree as an argument, it checks the cell count associated with that node. If the cell is zero, the tree with that node as a root node has not been labeled. The algorithm labels the tree and then returns the desired value. If the cell is nonzero, that value is returned.

```
procedure LABEL(node):
begin
    if count(node) = 0 then
        begin
            level ← 0;
            counter ← 1;
            count(node) ← 1;
            while level > 0 | SUBNODE do
                begin
                    if SUBNODE then
                        begin
                            node ← SUBNODEOF(node);
                            level ← level + 1;
                            counter ← 1
                        end
                    else
                        begin
```

```
                                    node ← PARENTOF(node);
                                    level ← level - 1;
                                    counter ← counter + count(node)
                                end;
                            count(node) ← counter
                        end
                    end;
            return count(node)
    end
```

If the cell count associated with the given node contains a zero, then the algorithm begins at this node, marks its count cell with a one and continues this process down through the tree until no subnode exists. This is accomplished by repeatedly taking the true branch on the test "if SUBNODE". The first statement of this branch selects a subnode of the current node by means of the procedure subnodeof and moves to it by assigning the returned value to node. The level is increased by one to record the movement down the tree; counter is set to one and the count cell of the new node is set to one. This records that there is at least one node in the subtree with this node as the root node. If any other nodes exist in this subtree, they will be determined by future processing of the algorithm. The count cell is used to accumulate the number of nodes of the subtree. This is accomplished by taking the false branch from the test "if SUBNODE". In this case, there are no subnodes and so the algorithm moves to the parent node. The level indicator is decremented by one to record the movement up the tree and counter is increased by the

number of subnodes already known to this node. Since the algorithm is moving up from a branch containing more subnodes, counter now contains the number of subnodes currently known to this node. If subnodes exist which have not been accounted for, the algorithm moves down to these nodes. If they have all been explored, the algorithm moves up to the parent node of this node leaving the number of nodes in that subtree in the cell count(node) and carrying this number in the variable counter to the next level. When the algorithm returns to the original node and no unexplored subtrees of this tree exist, then the number of nodes in the tree is contained in count(node) and this number is returned as the function value. As we did in Theorem 3, we use the Invariant Relation theorem to show that this algorithm labels the nodes correctly.

Theorem 4. Given the root of a tree and an array count as specified above, the function procedure LABEL labels an unlabeled tree and returns the number of nodes in that tree.

Proof. If the tree is labeled or if there is only one node, it is clear that the algorithm returns the correct value. Assume that the tree is unlabeled. An invariant relation R of the while loop is:

> The visited subtree having the current node as a root node is a labeled tree.

The condition C is: level > 0 | SUBNODE. Before the while loop begins, the visited subtree is the subtree consisting of the root node. Since the count cell for that node contains a one, it is labeled. After the loop halts, the condition R&¬C implies that the whole tree is the visited subtree, since level = 0 and SUBNODE is false, and that it is labeled.

It remains to be shown that R is an invariant relation of the loop and that the loop halts. Assume that R&C holds before entering the begin end block of the while loop. If a subnode exists, then move down to that subnode and set count(node) for that subnode to one. That node is now the visited subtree, it is labeled and the relation R holds. Assume that a subnode did not exist. By assumption R, the visited subtree with this subnode as a root node is labeled. Since there are no unvisited subnodes to this subtree, the whole subtree is labeled. Hence, both count(node) and counter contain the number of nodes in the subtree. Since C is true, level > 0, so we can proceed up to the parent node. The value which is contained in the cell count(node) for the parent node is the number of nodes in the visited subtree of the parent node just before the algorithm descended to an unvisited subnode. By adding count(node) to counter, counter contains the number of nodes in the new visited subtree. This is the subtree consisting of the old visited subtree union with the subtree with root node the current subnode. By setting counter into count(node) for the parent node, this subtree becomes a labeled subtree and the relation R holds. The while loop halts because there are only a finite number of nodes to be tested, once the algorithm selects a subnode as a new subnode it will not select it again and if no new subnode exists, level is reduced by one. Q.E.D.

Another version of the procedure LABEL is now given. We can see that this procedure is simpler if we allow recursive call in it. The proof of correctness of this procedure is ignored.

```
Procedure LABEL(node):
begin
    if count(node) = 0  then
       begin
           count(node) ← 1;
           while SUBNODE do
               count(node) ← count(node) + LABEL(SUBNODEOF(node));
       end
    return (count(node));
end
```

5. The Complexity of the Algorithm

We show that in the worst case the procedure BALANCE will locate a balance node of a spanning tree with n nodes in approximately $5n/2$ node visits and in the best case it will take approximately $n/2$ node visits. We first prove a lemma.

Lemma 2. The algorithm LABEL will label a tree with k nodes in exactly $2k - 1$ node visits.

Proof. The proof is by induction. If $k = 1$, then $2 \cdot 1 - 1 = 1$ and it is clear form the algorithm that one node is labeled in one node visit. Assume that the algorithm LABEL will label a tree of k nodes in $2k - 1$ node visits. Consider a tree of $k + 1$ nodes. If we remove any one leaf node, there remains a tree of k nodes, which by assumption can be labeled in $2k - 1$ node visits. Now rejoin the $(k + 1)$st node and consider the

extra visits needed to label the whole tree. One more visit is needed by the algorithm LABEL as it descends to this node on the branch SUBNODE = true. A second node visit is needed when the algorithm returns to its parent node after finding that SUBNODE = false. This is the only effect that this node will have on the actions of the algorithm. Therefore, the number of node visits is equal to $2k - 1 + 2 = 2(k + 1) - 1$. Q.E.D.

Theorem 5. Let T be a spanning tree with $n \geq 3$ nodes. The procedure BALANCE will locate a balance node of T in the worst case in $2(n - 2) + \lceil n/2 \rceil$* node visits and in the best case in $\lceil n/2 \rceil$ node visits.

Proof. Let u be the first node chosen as a candidate for a balance node of T. Let v_i, $1 \leq i \leq m$, be the adjacent nodes of u and assume each subtree T_i has n_i nodes. Note that $n = 1 + n_1 + \cdots + n_m$. The worst case will occur when the algorithm has to label each of the subtrees with root node v_i. This is the case whenever a leaf node of T is chosen because the remainder of T is the subtree of this leaf node. By Lemma 2, the number of node visits (N) to label every subtree with root node v_i is

$$N(m) = 1 + \sum_{i=1}^{m} (2n_i - 1) = 1 + 2 \cdot \sum_{i=1}^{m} n_i - m = 2(n - 1) - (m - 1).$$

The one accounts for first visiting the node u.

For fixed n, the function N(m) is a maximum when m = 1. In this case, $N(1) = 2(n - 1)$. After all of the nodes have been labeled, the greatest number of node visits needed to locate a balance node is $\lceil n/2 \rceil - 2$. This is because the while loop will halt when "tested" is greater than or equal to $n/2$ and at least the two nodes u and v have been tested by BALANCE already. Combining these two values gives the total number of nodes visits

*The notation $\lceil x \rceil$ means the ceiling function of x.

in the worst case:

$$2(n - 1) + \lceil n/2 \rceil - 2 = 2(n - 2) + \lceil n/2 \rceil .$$

The function N(m) is minimized when m = n - 1. That is, when each v_i is a root node of a tree with only one node. In this case N(n - 1) = n. Actually T looks like a spoked wheel with u at the center and the v_i, $1 \leq i \leq n - 1$, at the end of each spoke. In this case u is the balance node and the algorithm will halt after visiting only $\lceil n/2 \rceil$ nodes. This is clearly the best possible case because a minimum of $\lceil n/2 \rceil$ nodes must be visited before the algorithm will halt. Q.E.D.

References

[1] A. Aho, J. Hopcraft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[2] R. Conway, D. Gries, An Introduction to Programming, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1973.

[3] C. L. Liu, Introduction to Combinational Mathematics, McGraw-Hill, New York, 1969.

[4] R. Tarjan, Depth First Search and Linear Graph Algorithms, SIAM J. Computing 1, 2 (1972), pp. 146-160.