# Distributed Algorithms for Depth-First Search

S. A. M. Makki
Department of Computer Science
The University of Queensland
Queensland 4072
Australia
sam@cs.uq.oz.au

George Havas*
Department of Computer Science
The University of Queensland
Queensland 4072
Australia
havas@cs.uq.oz.au

**Abstract**

We present distributed algorithms for constructing a depth-first search tree for a communication network which are more efficient than previous methods. Our algorithms require $2|V| - 2$ messages and units of time in the worst case, where $|V|$ is the number of sites in the network, and as little as $|V|$ messages and time units in the best case. The actual number of messages and time units depends on the network topology and possibly on the routing chosen. We can interpret this to mean that the number of messages and time units is $|V|(1+r)$ in the worst case, where $0 \leq r < 1$ and the value of $r$ depends on the topology and the routing. In a best case for the simplest algorithm, for example when the underlying network has a ring topology, $r = 0$ and our basic algorithm requires $|V|$ messages and time units, regardless of routing. We extend the basic algorithm to achieve the same best case bound for other topologies. The worst case bound, which has $r = 1 - 2/|V|$, applies if the network topology is a tree. The improvement over the best of previous algorithms is achieved by dynamic backtracking, with a minor increase in message length.

**Keywords:** Distributed systems, depth first search, distributed algorithms, communication graph.

*Supported in part by the Australian Research Council

# 1 Introduction

Depth first search (DFS) has a wide variety of practical applications in distributed computing networks, such as finding the strongly connected components of a directed network or the biconnected components of a general network [2]. Strongly connected components can be used to detect directed cycles in a network, which may lead to deadlocks. Biconnected components may be used to check whether failure of one site may or may not disconnect a network [1]. A more efficient distributed algorithm for the DFS traversal of a network can help reduce the complexity of other distributed graph algorithms which use a distributed DFS traversal as their basic building block. Therefore it is desirable to be able to efficiently compute a depth-first search tree rooted at any arbitrary site of a network.

We present new algorithms which are more efficient than previous methods. First we present a model and give an overview of related research. We analyze the complexity of our basic algorithm and give some examples of performance on typical networks. We also describe some extensions to the basic algorithm and their effects.

# 2 The model

An interconnection network can be modeled by an undirected communication graph $G = (V, E)$ where $V$ and $E$ are the set of sites and the set of bidirectional communication links of the communication network, respectively. Sites are autonomous in that they perform their computation and communicate with each other only by sending messages. They do not share a common memory. Each site has a unique identity and has local information, such as the identity of each of its neighbors. Message delivery is handled by the communication subsystem which delivers a message from sender to receiver in finite time without any alteration or loss. A receiving site keeps messages in arbitrary order till they can be processed. Receiving and processing a message needs negligible time.

We evaluate the complexity of our algorithm using standard complexity measures. The communication complexity is the total number of messages sent during the execution of the algorithm. The time complexity is the maximum time elapsed from the beginning to the termination of the algorithm, assuming that delivering a message over a link requires at most one unit of time. (In spite of the fact that our messages have nonconstant length the assumption that messages take at most one time unit is reasonable since the time unit represents communication delays rather than actual data transmission time. The delays can be viewed as outweighing the data transmissions. The same assumption is made by Sharma *et al.* [11, 6] who use similar length messages.)

## 3 Related Research

Over the past decade a number of distributed DFS algorithms have been proposed [3, 1, 7, 4, 11, 6]. In principle, the algorithm of Sharma *et al.* [11, 6] simply distributes the traditional sequential recursive DFS over the network, allowing each site to handle communication with its parent and children. They claim that their distributed DFS algorithm uses $2|V|$ messages and $2|V|$ units of time. Their message bound improves on the message complexity of the algorithms of both Lakshmanan *et al.* [7] and of Cidon [4], which are $4|E| - (|V| - 1)$ and $3|E|$ respectively. The improvement is achieved by eliminating all real parallelism and putting more information in each message. The lack of parallelism is consistent with the result of Reif [10] which shows that the DFS problem is inherently sequential in a well-defined sense.

Table 1 presents the message and time complexity of previous distributed depth first search algorithms. The algorithm of Cheung [3] simply probes sequentially all the edges of the graph, and therefore requires $2|E|$ messages and time units. Awerbuch [1] improves the time complexity of Cheung's algorithm from $O(|E|)$ to $O(|V|)$ by using parallel message passing but at the cost of increasing the number of exchanged messages. Lakshmanan *et al.* and Cidon [7, 4] improve the message complexity of Awerbuch's algorithm by a constant factor through the elimination of some of the parallelism. Sharma *et al.* [11, 6] proposed algorithms which improved the time complexity of distributed depth-first search from $O(|E|)$ to $O(|V|)$ by removing more unnecessary parallelism from those earlier algorithms at the cost of increasing the message size. They allow the messages to carry more information, and thus they eliminate extra messages used in previous algorithms which were used by nodes to inform neighbors of their status.

| *Reference* | *Time complexity* | *Communication complexity* |
|---|---|---|
| Cheung [3] | $2\|E\|$ | $2\|E\|$ |
| Awerbuch [1] | $< 4\|V\|$ | $4\|E\|$ |
| Lakshmanan *et al* [7] | $2\|V\| - 2$ | $< 4\|E\| - (\|V\| - 1)$ |
| Cidon [4] | $\leq 2\|V\|$ | $\leq 3\|E\|$ |
| Sharma *et al* [11, 6] | $2\|V\|$ | $2\|V\|$ |

Table 1: Previous DDFS algorithm complexity

## 4 The algorithm

We use an essentially sequential algorithm. We use FORWARD and RETURN messages to explore the tree, which is similar to the method of Sharma *et al.* [11, 6]. Our key improvement is to reduce the number of RETURN messages by using dynamic

backtracking. We do this by including in our messages information which tells the receiver a potential return message address, which is not necessarily simply its parent.

Each node has a copy of the procedure Ddfs, for which pseudocode is given in Figure 1. The algorithm is sequential in the sense that at most one node may be executing its version of the algorithm at any one time. This makes the contents of the message in effect global.

We define a node to be a *split point* if, when visited, it has two or more unvisited neighbors or if it is the root node. Nodes which are not split points may be bypassed by RETURN messages.

Our messages comprise four components: message originator; message type (FORWARD or RETURN); the set *visited* of visited nodes; and *splitpoint*, the previous split point. In comparison with the message used in [11, 6], the only extra information is *splitpoint*. We use set theoretic notation for some of our data structures, avoiding consideration of nonessential complications associated with their implementation. Different implementations of sets may be used, as appropriate, with different ensuing message lengths. (Use of a bit array for the set *visited*, when appropriate, allows us to to store the message in $V + 2\log_2(V) + 1$ bits.) Likewise we do not consider here how to implement the procedure which selects the next element of a set.

A node which receives a message executes the Ddfs procedure. The only difference in message treatment is that FORWARD messages lead to execution of initialization code for local variables. Thus, in each case, a FORWARD message is issued to a neighbor (if there is an unvisited neighbor connected to this node), else a RETURN message is issued to an ancestor (if this node is not the root), else the algorithm terminates.

Our DDFS algorithm constructs the depth-first search tree for the distributed system in the order in which nodes are visited during the execution of the algorithm. The DFS tree is stored in a standard way: at termination the root node is informed and each node knows its parent and children in the tree, information which is stored in local memory. Each node has local variables *parent* and *childset* for storing the DFS-tree, plus a variable *unvisited* for storing its set of unvisited neighbors. The set *neighbors* of all neighbors of a node is available. The process starts with the root node which (in effect) gives itself a FORWARD message containing initial values of $\emptyset$ for *visited* and itself for *originator* and *splitpoint*. (This implicit message is not counted in the analysis.) We assume that no link or process failure occurs during execution of the algorithm.

## 5 Examples

It is easy to see that for a graph which is a simple cycle of length $|V|$ exactly $|V|$ messages are sent ($|V|-1$ FORWARD messages, 1 RETURN message). For a complete graph $2|V| - 3$ messages are sent ($|V| - 1$ FORWARD messages, $|V| - 2$ RETURN messages). In neither of these cases does the number of messages depend on the routing chosen. For analysis purposes, consider the graph in Figure 2.

```
procedure Ddfs; { executed on receipt of a FORWARD or RETURN message }
const neighbors, i;
var childset, parent, returnnode, unvisited, j; { local variables }
var messagetype, originator, visited, splitpoint; { message variables }
begin
    if messagetype = FORWARD then { node initialization }
        visited := visited ∪ {i}; { this node is visited }
        childset := ∅;
        parent := originator;
        returnnode := splitpoint; { save preferred return address }
    endif { initialization complete }
    unvisited := neighbors − (visited ∩ neighbors) ;
    if unvisited ≠ ∅ then { unvisited neighbors remain }
        j := Next(unvisited); { get next unvisited neighbor }
        unvisited := unvisited − {j}; { remove it from unvisited }
        childset := childset ∪ {j}; { add it to childset }
        if unvisited ≠ ∅ then { this node i is a split point }
            splitpoint := i; { we need to return to this node }
        else { this node i is not a split point }
            splitpoint := returnnode; { we will try to return to an ancestor }
        endif
        issue FORWARD message to node j;
    else { time to backtrack }
        if i = parent then stop { i is the root }
        else { we need to backtrack }
            if returnnode ∈ neighbors then { appropriate ancestor is a neighbor }
                issue RETURN message to node returnnode;
            else { previous split point was not adjacent }
                issue RETURN message to node parent;
            endif
        endif
    endif
end
```

Figure 1: Pseudocode of the algorithm for node $i$

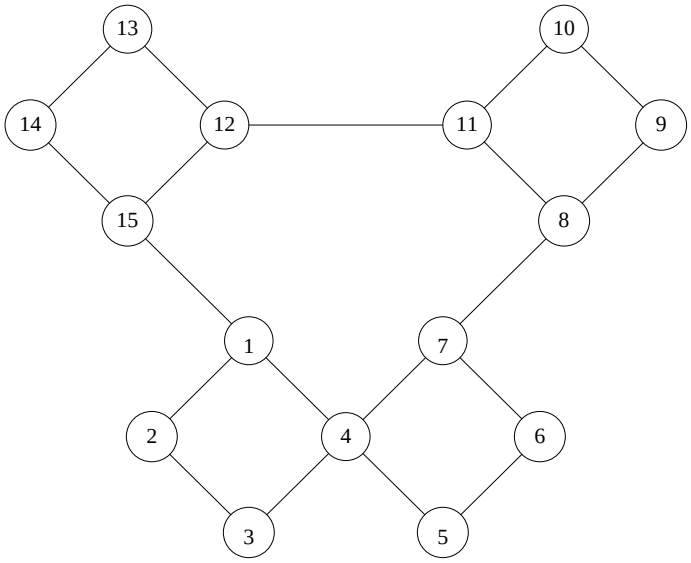


Figure 2: A sample graph

First consider node 1 as root. The total number of messages depends very much on the routing chosen. We describe routings by the order in which nodes receive FORWARD messages. In all cases the number of FORWARD messages is 14 ($= |V| - 1$). If, for example, routing $\langle$ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 $\rangle$ or $\langle$ 1, 4, 7, 8, 11, 12, 15, 14, 13, 10, 9, 6, 5, 3, 2 $\rangle$ is chosen, the number of RETURN messages is 6. If the routing is $\langle$ 1, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 $\rangle$ the number of RETURN messages is 8. A worst case of 14 RETURN messages arises from routing $\langle$ 1, 15, 12, 13, 14, 11, 8, 9, 10, 7, 4, 5, 6, 3, 2 $\rangle$.

Choosing node 4 as root is easier to analyze due to symmetry. In this case the number of RETURN messages varies between 8 (achieved by $\langle$ 4, 3, 2, 1, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5 $\rangle$, for example) and 14 (achieved by $\langle$ 4, 1, 2, 3, 15, 12, 13, 14, 11, 8, 9, 10, 7, 6, 5 $\rangle$). If each routing decision is equally possible at each node then there is an average of 9.5 RETURN messages.

Algorithms described in this paper are empirically studied in detail in [9].

# 6 Analysis

Correctness of the algorithm follows immediately from correctness proofs for the standard algorithms for nondistributed systems. Our algorithm is a distributed variant of the nondistributed algorithm. During one execution of the procedure, each node sends (at most) one message. It does this as its final executable statement. Hence at most one node is executing the algorithm at any one time, that is, in node execution terms it is sequential.

A best possible algorithm for DDFS has message complexity at least $|V|$, since each nonroot node must receive a FORWARD-type message and the root node must

receive a RETURN-type message. In the worst case, for a tree network, each FORWARD message must be traced back by a RETURN message, since no alternative route back is available. In such cases the total number of messages is $2|V| - 2$.

**Theorem:** The message and time complexity of this DDFS algorithm is between $|V|$ and $2|V| - 2$. (This can be expressed as $|V|(1 + r)$, where $0 \leq r < 1$, and $r$ depends on the topology and routing.)

**Proof:** The message complexity is the total number of FORWARD and RETURN messages. The number of FORWARD messages is $|V| - 1$, because each nonroot node has exactly one FORWARD message sent to it. The number of RETURN messages is at least one, since a RETURN message must be received by the root. The number of RETURN messages is at most $|V| - 1$, because each nonroot node sends at most one RETURN message, possibly none. (When a node sends a RETURN message it is visited itself and has no unvisited neighbors, so the node in question cannot receive any further messages.) Thus the total number of messages is between $|V|$ and $2|V| - 2$. Since the algorithm is sequential, the message and time complexities are the same.

# 7 Extensions

There are some natural extensions to this algorithm suitable for practical networks. In each case we need to send longer messages.

Observe that there is some inefficient behaviour in some of the examples presented. Thus, for a complete graph with three or more vertices the basic algorithm saves just one RETURN message (the last node visited sends its RETURN message to its grandparent rather than its parent). However the last node visited is adjacent to the root, and if it knew the root and that all nodes were visited it could send its RETURN message directly to the root, using one instead of $|V| - 2$ RETURN messages.

We can achieve this quite readily: extend the message to include a variable *root* and a set *knownunvisited*, with the natural syntax and semantics. The details are straightforward. Variable *root* is initialized by the root node and never changed. The set *knownunvisited* is initialized by the root node to $\emptyset$. It is updated in the procedure body ($knownunvisited := knownunvisited \cup unvisited$) after (the local) *unvisited* is updated. Before sending a FORWARD message to node $j$, we remove $j$ from *knownunvisited*. Then, before sending a RETURN message, we check if *knownunvisited* is empty. If so, check if *root* is adjacent, and, if so, send the RETURN message directly to *root*. The cost is messages with length about double the previous size, depending on set representation used.

With this modification complete graphs are handled with $|V|$ messages regardless of routing, which is best possible. Both worst case routings presented for the sample 15 node graph are handled with 7 RETURN messages instead of 14. The two cases previously with 6 RETURN messages are now handled very differently: the former needs only one RETURN message while the latter still requires 6. Both presented routings with 8 RETURN messages now require just one.

Alternative extensions arise if the number of nodes in the graph is known in advance or if shortest path information is known. Thus, if the number of nodes is known it suffices to keep a count rather than the set *knownunvisited*, reducing the message length. Shortest paths can be used to reach nonadjacent return addresses, by adding an extra field to RETURN messages to indicate indirect return.

# 8 Conclusions

We have presented distributed algorithms for constructing a depth-first search tree in a communication network which are more efficient than previous methods. Our algorithms require $2|V| - 2$ messages and units of time in the worst case, where $|V|$ is the number of sites in the network, and as little as $|V|$ messages and time units in the best case. Thus the algorithms have message and time complexity $|V|(1 + r)$ in the worst case, where $0 \leq r < 1$ . Improvement over previous methods comes from dynamic backtracking and elimination of transit nodes for RETURN messages, at the expense of a small increase in message length. Good network structures for our algorithm include biconnected graphs or graphs with biconnected components. Since the execution of the algorithm is sequential it is suitable for both synchronous and asynchronous networks.

The algorithms described in this paper provide the basis of fault tolerant algorithms for reconstructing a distributed depth-first search tree in an interconnected communication network in a dynamic environment [8]. (A fault tolerant distributed depth-first search tree algorithm for a simpler model of computation appears in [5].)

# References


[1] B. Awerbuch, A new distributed depth-first search algorithm, *Inform. Process. Lett.* **20** (1985) 147–150.

[2] E. J. H. Chang, Echo algorithm: Depth parallel operations on general graphs, *IEEE Trans. Software Eng.* **8** (1982) 391–401.

[3] T. Cheung, Graph traversal techniques and the maximum flow problem in distributed computation, *IEEE Trans. Software Eng.* **9** (1983) 504–512.

[4] I. Cidon, Yet another distributed depth-first-search algorithm, *Inform. Process. Lett.* **26** (1987/88) 301–305.

[5] Z. Collin and S. Dolev, Self-stabilizing depth-first search, *Inform. Process. Lett.* **49** (1994) 297–301.

[6] D. Kumar, S. S. Iyengar and M. B. Sharma, Corrections to a distributed depth-first search algorithm, *Inform. Process. Lett.* **35** (1990) 55–56.

[7] K. B. Lakshmanan, N. Meenakshi and K. Thulasiraman, A time-optimal message-efficient distributed algorithm for depth-first-search, *Inform. Process. Lett.* **25** (1987) 103–109.

[8] S. A. M. Makki and G. Havas, Reconstructing a distributed depth-first-search tree after network topology changes, *Proc. Sixth IASTED/ISMM Internat. Conf. Parallel and Distributed Computing and Systems*, 335–337, Acta Press, Anaheim (1994).

[9] S. A. M. Makki and G. Havas, Empirical analysis of distributed depth-first search algorithms, *Proc. Eighth IASTED Internat. Conf. Parallel and Distributed Computing and Systems*, to appear (1996).

[10] J. H. Reif, Depth-first search is inherently sequential, *Inform. Process. Lett.* **20** (1985) 229–234.

[11] M. B. Sharma, S. S. Iyengar and N. K. Mandyam, An efficient distributed depth-first-search algorithm, *Inform. Process. Lett.* **32** (1989) 183–186.