An Efficient Method for Constructing a Distributed Depth-First Search Tree

S. A. M. Makki and George Havas*
School of Information Technology
The University of Queensland
Queensland 4072 Australia
sam@it.uq.oz.au havas@it.uq.oz.au

Abstract

We present an efficient distributed depth-first search algorithm. The algorithm in the worst case requires $(1+r)(|V|-1)$ messages and $(1+r)(|V|-1)$ units of time, where $0 \leq r < 1$ and $|V|$ is the number of sites in the network. The value of r depends on the search path and the graph topology. In the best case when $r = 0$, the algorithm requires only $|V|-1$ messages and time units. This new algorithm improves over the best of previous algorithms by using a stack-type structure which enables better dynamic backtracking. The improvement in worst case complexity is only minor. However it is much better in practice, manifested by significantly smaller values of r.

Keywords: Distributed systems, Depth-First-Search, distributed algorithms, communication graph.

1 Introduction

Depth-first search (DFS) is a fundamental operation in graph traversal. It is a principal constituent of other graph algorithms, such as finding the biconnected components in a directed graph and the strongly connected components of a general graph [10]. 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.

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* [9, 5], who use similar length messages.)

2 Previous algorithms

The earliest distributed depth-first search algorithm due to Cheung [2] sequentially probes all the edges of the underlying network graph, and thus requires $2|E|$ messages and time units. Awerbuch [1] improves the time complexity of Cheung 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 [6, 3] improve the message complexity of Awerbuch's algorithm by a constant factor through the elimination of some of the unnecessary parallelism. Sharma *et al* [9, 5] propose algorithms which improve the time complexity of those earlier algorithms to $2|V|$ by removing extra parallelism at the cost of increasing the message size. They allow the messages to carry more information, and therefore they

*Supported in part by the Australian Research Council

are able to eliminate some messages used in previous algorithms which were mainly used by nodes to inform neighbors of their status.

We present and analyse improved distributed depth-first search algorithms based on dynamic backtracking [7, 8] which reduce the time and message complexity to $(1 + r)|V|$ (where $0 \leq r < 1$ and depends on the traversal path and the network topology). We use two types of messages (FORWARD and RETURN) for handling communication between the nodes. The dynamic backtracking eliminates the need for eventually following each FORWARD message with a RETURN message in the backtracking stage of the search. In order to achieve this, we simply include in our messages information which tells the receiver a potential return message address, which is not necessarily its parent. Thus the RETURN messages are able to by-pass some intermediate nodes and as a result reduce the number of RETURN messages. Although these algorithms show better results for almost all network topologies compared with their previous counterparts, experimental results [8] reveal less than desired performance shown as relatively high values for r in practice. This is because, as the search progresses, previous information regarding return addresses stored at visited nodes is no longer accessible. (In a distributed system, nodes do not have global information of the topology or the search progress status at each stage.) This lack of information removes the possibility of dynamically updating return addresses. It reduces their potential effectiveness when nodes would have been able to select a better return address if they had been able to update their old data.

3 The new algorithm

In this section we present a new algorithm which overcomes almost all of the inefficiencies of the previous algorithms. Again we use the same two types of messages (FORWARD and RETURN) in order to explore the network graph. The key improvement is to reduce the number of RETURN messages further by providing the latest global search information for the visited nodes at each stage of the search. This is done by using a stack-type structure, since stacks inherently satisfy the DFS property. This stack-type structure, which we call a stack for brevity, allows all standard stack operations plus some extra operations. Thus we allow additions to the stack of sets of elements (with the same effect as pushing every element of the set onto the stack in arbitrary order). Also we allow inspection of the entire stack and deletion of any specified element from the stack. The end result is that visited nodes have greater access to the latest search information and this enables them to compute better return addresses. Thus the content of the stack provides information which makes possible use of a shorter return path. The stack (implicitly) includes information giving the parent of the previous unvisited node (which is the ultimate return address). This enhances the selection of an appropriate return address.

Each message (FORWARD or RETURN) contains four components: message type; sender; destination; and the stack (which at each stage contains the visited nodes and their unvisited neighbors in the search path). Each node on the stack also is accompanied by a flag, which indicates the node's status (visited or unvisited).

A node when first visited puts all unvisited neighbors on the stack after deleting them if they appear earlier in the stack. Then it sends a FORWARD message accompanied by the stack to the next unvisited neighbor. This neighbor is usually the node at the top of the stack. However, when the search reaches a node which does not have any unvisited neighbors we have to backtrack. In this case the stack is searched for the first unvisited node (from top of the stack), if any.

The process starts from the *root*, which puts itself as the first visited node and its neighbors as unvisited nodes on the stack. Then it selects its top of stack neighbor to send a FORWARD message. When a node receives a FORWARD message, it changes its status to *visited*. Then it removes all of its *unvisited* neighbors from the stack (if they are already there) and places them on top of the stack. It does so because at each stage of the search a node must visit all its descendants before starting to backtrack in order to satisfy the DFS property. To continue the search it selects its top of stack unvisited neighbor, and so on. If a node does not have any unvisited neighbors it checks from the top of the stack for the first unvisited node. In order to reach this selected node the search backtracks via as good as possible a path to the parent of the first unvisited node.

In the algorithms of [7] backtracking is strictly confined to a predefined return path involving either split points or direct parents. Here we have better options. We choose to do a greedy selection process which very effectively backtracks in practical situations.

The process continues until no unvisited nodes remain on the stack, when the algorithm terminates.

Each node has a copy of the procedure *stackdfs*, for which pseudocode is given in Figure 1. This algorithm, like the previous algorithms in [7], is sequential. With appropriate data representations the size of the message does not differ greatly from that of the messages used in [7]. Messages in the new algorithm contain a

stack of nodes instead of a set. If a sequence of node identifiers is used to represent both the stack and set, then the stack and set have the same size, which in the worst case can be $|V|\lceil \log |V| \rceil$ bits. The new algorithm also includes an extra status bit with each node in the stack so, counting that bit, in the worst case the message needs $|V|$ extra bits, while there is a saving of $\lceil \log |V| \rceil$ since the *splitpoint* of the earlier algorithms is no longer needed. Thus there is only a minor increase in message size.

This algorithm constructs the depth-first search tree in a distributed environment in the order in which the nodes are visited during the execution of the algorithm. At the termination of the algorithm each node knows its parent and its set of children, which is stored in its local memory. Each node has local variables which are initialised when a node receives a FORWARD message. The process begins from the root node which gives itself a FORWARD message (not counted in the analysis) containing initial values of (*root*, visited) on the *stack* and itself for *originator*. It is assumed that no link or process failure occurs during execution of the algorithm.

4 Examples

This algorithm outperforms its previous counterparts. For example for a cycle of length $|V|$ it requires $(|V| - 1)$ messages which is the number of FORWARD messages, but no RETURN messages are required. Also, for a complete graph the total number of messages required is $(|V| - 1)$, since to visit all nodes $(|V| - 1)$ FORWARD messages are required and again no RETURN messages are needed. This algorithm also works better on many different tree structures. In the worst case it requires $2|V| - 3$ messages. (This situation arises when the tree has only two branches, one branch contains only one node and the search first traverses the longer branch.)

5 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.

An optimal algorithm for distributed depth first search has message complexity at least $|V| - 1$ since, in order to visit each node, all non-root nodes must receive a FORWARD message. In the worst case, the algo-

```
procedure stackdfs; {executed on message receipt}

{We use a natural functional notation to describe
the operations on the stack, avoiding consideration of
nonessential complications associated with their imple-
mentation. Different implementations may be used, as
appropriate, with possibly different ensuing message
lengths. The stack contains both nodes and flags. We
denote by stack.node the sequence of nodes in the stack
and for n ∈ stack.node we denote the associated sta-
tus flag by stack.flag_n. We do not detail here how
to implement the operations. However they can all
be readily implemented satisfying the assumption that
local computing time is negligible compared with com-
munication time. The SelectVisited() operation is par-
ticularly interesting: our implementation uses a greedy
selection method which performs very well in practice.}

const neighbors, i; {local variables}
var childset, parent, returnnode, n; {local vars}
var messagetype, originator, stack; {message vars}
begin
  if messagetype = FORWARD then {initialization}
    parent := originator;
    childset := ∅;
    stack.flag_i := visited;
    ∀n ∈ neighbors do
      if n ∈ stack.node then
        if stack.flag_n = unvisited then
          Remove(n, stack);
          Push( (n, unvisited), stack );
        endif
      else
        Push( (n, unvisited), stack );
      endif
    end∀
  endif {initialization complete}
  if ∃n ∈ neighbors | stack.flag_n = unvisited then
    SelectUnvisited(n, stack);
    childset := childset ∪ {n}; {add it to childset}
    issue FORWARD message to node n;
  elseif {time to backtrack or terminate}
    ∃n ∈ stack.node | stack.flag_n = unvisited then
    SelectVisited(returnnode, stack); {for backtrack}
    issue RETURN message to node returnnode;
  else {no unvisited nodes remain}
    Stop(); {the DDFS tree is complete}
  endif
end
```

Figure 1: Pseudocode of the algorithm for node i

rithm must backtrack from visited nodes which do not have an unvisited neighbor to previous visited nodes.

Theorem: The message and time complexity of this algorithm is between $|V|-1$ and $2|V|-3$. (This can be expressed as $(|V|-1)(1+r)$, 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 non-root node receives exactly one FORWARD message. Unlike earlier methods it is possible that no RETURN messages are required (witness ring and complete graph topologies). The number of RETURN messages is at most $(|V|-2)$, because in the worst case the search may need to backtrack from all the visited (non-root) nodes except one, giving up to $(|V|-2)$ messages (witness the worst case tree example). Therefore, for the best and worst cases the total number of FORWARD and RETURN messages is between $(|V|-1)$ and $(2|V|-3)$ respectively. This can be expressed as $(|V|-1)(1+r)$, where $0 \leq r < 1$. The value of r depends on the network topology and on the routing chosen. Since the algorithm is sequential, the message and time complexities are the same.

6 Experimental results

An experimental method for evaluating distributed DFS algorithms is outlined in [8]. We follow the same approach here. Thus to show the behavior of the algorithm on different network structures we have carried out a reasonable number of tests. In order to improve the quality of our results we have taken advantage of symmetry when appropriate.

We consider a number of the standard graph structures of various sizes as used in distributed systems. In our experiments we counted the actual number of RETURN messages for each of 1200 randomly constructed dfs trees. Since the number of RETURN messages (which we denote by $\#R$) can range from 0 to $|V|-2$ for this algorithm, we also use a ratio ρ) (related to r) which ranges from 0 to 1 as a more uniform metric of performance. Thus we define $\rho = \#R/(|V|-2)$. Best possible performance has $\rho = 0$, worst possible performance has $\rho = 1$. (This ρ is defined slightly differently to that in [8], but they are comparable.)

The table in figure 2 compares the average value of ρ derived from 1200 tests conducted on hypercube structures using the new algorithm with the results for the basic and extended algorithms presented in [8]. (The new algorithm is designated *Stack ddfs* in these and subsequent plots.)

For mesh and torus type networks the new algorithm shows impressive improvements compared to the basic and extended ddfs algorithms. The results for mesh and torus type networks are shown in Figure 3 and Figure 4 respectively.

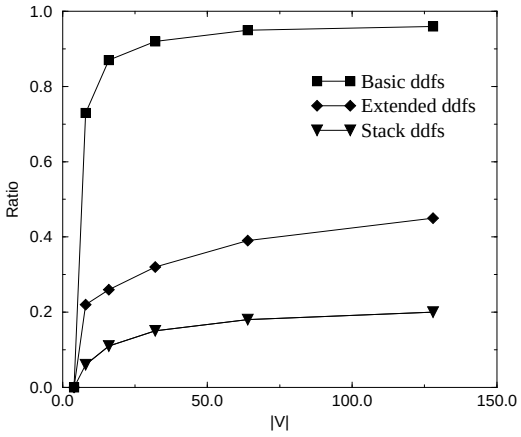

Figure 2: Performance on hypercubes

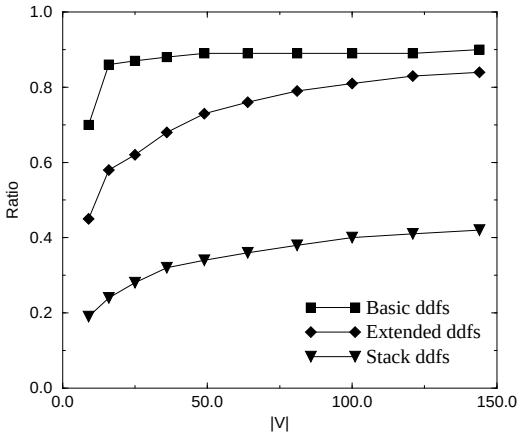

Figure 3: Performance on meshes

Finally we compare the performance of the stack ddfs algorithm on the three different types of topologies in Figure 5.

The results of these experiments clearly illustrate that this new algorithm outperforms the previous algorithms. This is achieved as a result of exploitation of the available information at each stage leading to better practical values for the parameter r.

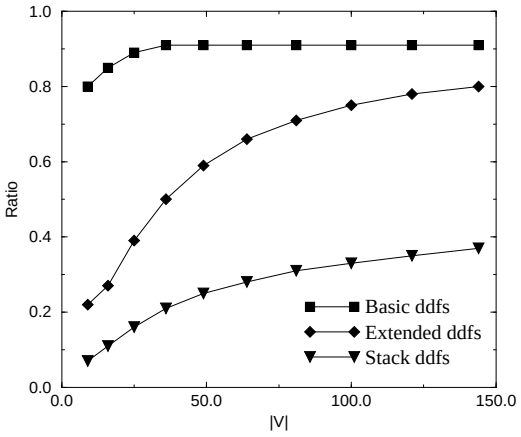

Figure 4: Performance on toruses

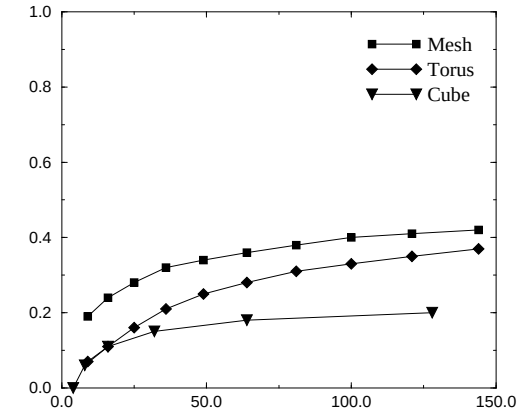

Figure 5: Stack ddfs topology comparison

7 Conclusion

We have presented a more efficient distributed depth-first search algorithm which constructs a depth-first search tree in a communication network. The algorithm has message and time complexities of $(|V| - 1)(1+r)$ in the worst case, where $0 \leq r < 1$. Improvement over the previous algorithms is achieved from better dynamic backtracking and elimination of transit nodes for RETURN messages, at the expense of a small increase in message length. Furthermore, unlike earlier algorithms, there is no need for a RETURN message to go all the way back to the root node.

As the experimental results show, the number of worst cases are reduced significantly in practice. The best result for this algorithm is achieved when the underlying network topology has a Hamiltonian path and the depth-first search traversal follows such a path. The experiments show that the algorithm has very good performance on dense graphs as well as on sparse graphs. This is because in dense topologies the increase in connectivity results in a dramatic shortening of the return paths. (Indeed, it increases the possibility of existence of Hamiltonian paths, see Harary [4, p. 66].) Also note that, since the execution of the algorithm is sequential, it is suitable for both synchronous and asynchronous networks.

References

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

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

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

[4] F. Harary. Graph Theory. *Addison-Wesley* Publishing Company, 1969.

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

[6] 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.

[7] S. A. M. Makki and G. Havas, Distributed algorithms for Depth-First Search, *Inform. Process. Lett.* **60** (1996) 7–12.

[8] 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*, Acta Press (1996) 292–296.

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

[10] R. E. Tarjan, Depth first search and linear graph algorithms, *SIAM J. Computing* **1** (1972) 146–160.