COOPERATIVE DISTRIBUTED ALGORITHMS
FOR DYNAMIC CYCLE PREVENTION

by

Shmuel Katz and Oded Shmueli

Technical Report #330
July 1984
Index terms

Distributed algorithms, program development, dynamic cycle prevention, graph searches, distributed invariants, coopération among operations
Cooperative Distributed Algorithms for Dynamic Cycle Prevention

Shmuel Katz and Oded Shmueli
Computer Science Department
The Technion, Haifa, Israel

ABSTRACT

Parallel distributed algorithms are presented for adding and deleting arcs in a directed graph without creating a cycle. Such algorithms are useful for a variety of problems in distributed systems such as deadlock prevention. The algorithms operate in a realistic asynchronous computer network environment. In this environment there are numerous possible interactions between instances of the algorithm at various nodes.

The distributed algorithms are derived from a sequential algorithm. In developing the distributed version of the algorithm from a sequential version, the vital role of an invariant is emphasized. Global correctness of the distributed algorithms relies on (locally) preserving this invariant. Interactions and cooperation between various activations of the algorithms are exploited in order to minimize redundant computation.

C.R. Categories: C.2.4 Distributed systems; D.1.3 Concurrent programming
1. The Problem

Parallel distributed algorithms are given for adding and deleting arcs in a distributed directed graph without creating a cycle. The algorithms operate in a realistic asynchronous computer network environment. In this environment there is a large number of possible interactions between instances of the algorithm at various nodes (sites). Such algorithms are useful in solving system problems such as distributed deadlock prevention and distributed concurrency control [2,4].

The algorithms are systematically developed from global, sequential algorithms relying on the same invariant to guarantee correctness. In the sequential algorithms the invariant holds "before" and "after" each operation. In the distributed version, the invariant has to hold at all times. Otherwise, operations which proceed in parallel may be "misled" by temporary inconsistencies.

The changes which must be made to the original algorithms in order to allow parallel execution, to completely eliminate global information, and to otherwise adjust to a message-passing environment, are analyzed. Interactions and cooperation between various activations of the algorithms are exploited in order to minimize redundant computation.

A directed graph G is acyclic iff each vertex v of G can be associated with a number n(v), so that if there is an arc from i to j then n(i) < n(j). Technically, following [8], n(v) is maintained as a 2-tuple (round, priority). Intuitively, round number r means that the node was numbered in the r'th numbering, while the priority indicates the order in which nodes were numbered with r.

A node numbered graph G is conservative if whenever there is an arc from a node i to a node j in G then the condition INV holds:

\[(INV) \ (\text{round}_i < \text{round}_j) \text{ or } [(\text{round}_i = \text{round}_j) \text{ and } (\text{priority}_i > \text{priority}_j)].\]

Clearly, a conservative graph is acyclic. INV contains the term \((\text{priority}_i > \text{priority}_j)\) due to the manner in which graph searches are employed.

* We use standard graph theory notation [1,6].
to achieve a correct node numbering. As will become clear later, the inverse ordering is used because the number of nodes which will be marked with a given round number is not known in advance. A similar numbering approach appears in [6].

Suppose the arc from $i$ to $j$ is to be added to a conservative graph $G$. If INV holds between $i$ and $j$, then the arc may be immediately added, since the graph remains conservative. Otherwise, we do not know whether or not there is a path from $j$ to $i$; a renumbering (i.e., a re-marking) may be attempted in order to satisfy INV and allow the arc’s addition. In the sequential versions two methods may be used for renumbering, based on depth first search (dfs) and breadth first search (bfs) [1].

In the dfs sequential method, a new round number larger than all the previous round numbers (including $i$'s) is chosen, and a depth first search from $j$ is started. In this search each node encountered will be assigned the new round number. The priority of a node will be equal to the number of nodes already completely visited in the search; thus the priority is assigned as the (recursive) dfs call terminates operation at the node. When the search is completed, INV is checked again. Now, there is no path from $j$ to $i$ iff INV is true.

The bfs method uses a similar technique. One important difference between the methods is the manner in which bfs assigns priority numbers. In bfs, the priority indicates “longest path to the furthest sink node” (a sink is a node with no outgoing edges). This may lead to having nodes with the same (round, priority) values.

The worst case complexity of the bfs and dfs approaches in a centralized sequential environment is identical. However, assuming all edge addition requests are equally likely, the dfs approach is superior, because there is a total ordering among the nodes. It is shown in [8] that under the above assumption half of the edge addition requests may be handled without initiating a search. A similar estimate for the bfs method is unknown.
2. Options in Developing a Distributed Algorithm

In the distributed environment graph nodes are considered as processes communicating by message passing. Each process has local memory and a message queue which is processed in FIFO order. Node i will contain the information that it is connected by arcs to nodes \( j_1, j_2, \ldots \) (but i does not know which nodes are connected to it). The number of nodes, \( N \), is assumed fixed (but need not be known at any node). Any two processes can communicate, regardless of whether they are connected by a graph arc, and the graph should be considered as a data structure maintained distributedly by the processes. It is, however, true that while re-marking (renumbering) nodes, messages usually (but not always) will be sent between nodes connected via an edge in the graph.

We assume that for each potential arc \((i,j)\) there is at most one request active at any time to add that arc. Furthermore, arcs are not requested while already in the graph. These technical restrictions can be removed at the cost of somewhat complicating the algorithms.

The possibility of concurrent requests leads us to the first solution, which is an intermediate stage between the sequential and distributed algorithms. A central control node is chosen. All edge addition requests are sent to the control node which processes them one at a time. Deletions may be processed locally by simply removing the edge. The control node employs either the bfs or the dfs method, where the search is conducted by message passing. In this setting the advantage of using the distributed bfs is that the search operates in parallel and hence may be of a short duration. Still, the dfs approach has the advantage of initiating fewer searches.

The central control node solution may exploit parallelism as follows. After initiating the (primary) search, it may initiate additional (secondary) searches with smaller round numbers. When two searches "collide" at a node, the one with the smaller round number is "suspended". The details of how such encounters are handled will be discussed in a subsequent section. Basically, the primary search is
assured of termination and of being able to conclusively check INV following its termination.

Secondary searches may terminate inconclusively. In this case they are again eligible to become primary searches. The reasons for a possible inconclusive termination are described in the next section. An obvious tradeoff exists between overhead, in terms of number of messages, and the level of concurrent searching, i.e. the number of secondary searches which operate concurrently with the primary search. However, note that the primary search will proceed exactly as if it were alone. The above discussion holds for both the bfs and dfs approaches.

The central node solution has some drawbacks. The central node may become a communication bottle-neck due to the large volume of messages. The queueing order is artificial, requests that imply non-overlapping searches are scheduled in the artificial order of arrival at the central node. Trying to resolve these problems, we arrive at a totally distributed solution. Because the solution is distributed, there are no primary searches. Hence, it may happen that no search will end up with a conclusive answer. This is likely only under high request load. Should a conclusive answer be essential, a mechanism called definitive search is provided which guarantees a conclusive answer.

Below, the dfs distributed version is described, ignoring for the moment arc deletions and definitive searches. The bfs version is seen in Section 4, and three correctness theorems are proven in Section 5. The impact of deletions is considered in Section 6, while the limited freezing needed for definitive search is described in Section 7.

3. A Distributed Algorithm (dfs version)

Given the global sequential dfs solution, and the nature of the distributed environment, some of the properties of the distributed algorithm are immediate. First, a request to add an arc from i to j should be directed to i, where the checks
of the invariant should be performed, and the actual addition of the arc is done (should this be safe). This is necessary so that there is no danger of \textit{round}_i increasing between the check and the actual addition of the arc, possibly leading to a violation of the invariant.

In the distributed version, no process has global knowledge of the current largest round number, and in any case new searches definitely can begin elsewhere in the graph while previous ones are still going on. Thus the round number of a search can not be guaranteed to be the largest in the graph. However, note that when \( i \) is initiating a search at \( j \) due to a request to add the edge from \( i \) to \( j \), it is sufficient to choose any value larger than \( \text{round}_i \). In order to guarantee that the round numbers chosen for different inspections be distinct, each round number is a 2-tuple whose first component is an integer and its second (less significant) component is a (unique) node id.

Finally, since numerous requests will be processed in parallel, the invariant must be maintained not only before or after the searches, but also \textit{during} their execution. The depth first search will be initiated from \( j \), and clearly messages must be sent in turn along all arcs emanating from \( j \) in the graph, 'activating' the search there. It remains to decide what should be maintained at each node in order to conduct such searches, and what should occur when such a search request reaches a node \( k \). The node \( k \) clearly must maintain its round and priority numbers (referred to in the sequel as the \textit{established} values), as well as information about searches which are going on through \( k \), but have not yet been completed.

A node will receive a structured message activating a search with a tentative round number \( r \), a tentative priority \( p \), and the source of the message, \( s \). Several situations are possible and must be treated:

(1) a search message first reaches node \( k \) and the established round number, \( \text{estr}(k) \), is larger than the round number of the search, \( r \). This means that in any continuation of the search from \( k \), all the nodes encountered have
been marked with round numbers larger or equal to estr(k) (and thus to r). It is forbidden to change the round numbers of those nodes to r, since then the invariant might be violated (if a node with a larger round number should now be connected to a node newly marked with a smaller r). Thus there is no reason to continue the re-marking from k with round r. The search should simply "retreat" to node s, and continue the search from there. Note that in this situation, if there should indeed be a path from j to i which passes through k, i will have a larger round number than j, and INV will not hold when the search terminates.

(2) a search reaches k with estr(k)<r. This does not necessarily mean that the new search can immediately continue, since another search, also ongoing through k, may have a round number r'>r. In order to easily check whether this is the case, each node k will also maintain tenr(k) (tentative round number) which is the value of the largest round number in a search ongoing through k. Three situations must be considered:

(2a) estr(k)<r and r<tenr(k). By reasoning similar to case (1), continuing the search with r is a waste of effort, since all of the descendants of k will ultimately be marked with a larger round number. However, in this situation, the search activated by the message with parameters s, r, and p, cannot simply retreat to s and continue, since not all of the descendants of k have already been marked by tenr(k). Thus the new search should simply wait until it can safely retreat.

(2b) estr(k)<r and r=tenr(k). This means that the node k has already been passed in this search but that the search has not yet retreated from k, and yet a new search message with the same round number is received. However, such a situation cannot ever occur in a dfs because the graph is always cycle-free! Only when the response following a search retreats to k after having remarked a subgraph reachable from one of the neighbors of k, can a search ongoing through k again reach that node.
(2c) \( estr(k) < r \) and \( \tau > tenr(k) \). Now the search \((s,r,p)\) should "take over" the task of marking the descendants of \( k \). The variable \( tenr(k) \) is to be updated to the value of \( r \), and the search should be continued at one of the sons of \( k \).

For each arc from \( k \) to \( l \), we now consider the maximal round number of searches which have returned along that arc to \( k \), maintain that value in a variable \( \text{largest}[l] \) in the node \( k \), and define \( estr(k) \) to be the minimum of those values. In this way it is guaranteed to be no larger than the established round numbers of its neighbors, as required. Of course, the established priority number, \( estp(k) \), will have to be maintained so that the invariant is correct even when the round numbers of neighbors are equal.

When a search returns along an arc \( l \), it may update the largest value of the round for that arc, possibly leading to a recomputation of \( estr(k) \) and the priority. Then all searches waiting at \( k \), including the one which has just returned, should be compared with \( estr(k) \). All those with round numbers less than or equal to \( estr(k) \) may resume operation by sending an answer message to the source of their search message, and thus cease waiting at \( k \).

Note that some continuations of searches might return along an arc to \( k \) even after the search itself has been preempted by another search, or even returned to its origin \( s \). In that case, the round number will be less than the largest round number on that arc (otherwise \( estr(k) > r \) could not have been true) and thus the returning answer message should have no effect and is simply ignored.

In order to guarantee that progress is made, whenever a new search message is to be sent from \( k \), the next arc to be searched will be the one with the smallest round numbers in descendants, i.e., the arc with \( \text{largest}[l] = estr(k) \).

Consider a search resulting from a request for adding the edge from node \( i \) to node \( j \). Upon search return to the initiating node \( i \), the invariant is checked. If it holds then the edge is added. If \( i \)'s round number equals the search round number, then there was a way to reach node \( i \) from node \( j \) and the edge addition request is denied. Otherwise, the search terminates inconclusively.
Each node must potentially fulfill a role (1) as the receiver of a request to add an arc, (2) as the origin of a search (which is the target of the arc to be added), (3) and as a node encountered during a remarking. The messages received at a node \( k \) are:

- **ADDARC\((j)\)**: a user-generated request to \( k \) to add the arc \((k,j)\) to the graph.
- **GETVAL\((j)\)**: a request from \( j \) to \( k \) for the established values of \( \text{round}_k \) and \( \text{priority}_k \).
- **RETURNVAL\((j,r,p)\)**: a response from \( j \) to a previous **GETVAL\((k)\)** request.
- **SEARCH\((j,r,p)\)**: a request from \( j \) for a remarking with round number \( r \) and a priority so far of \( p \).
- **ANSWER\((j,r,p)\)**: a response to a previous **SEARCH** request which indicates that \( j \) has a round number of at least \( r \) and that \( p \) nodes have been assigned a round number of \( r \).

The precise variables at a node and the algorithm described in terms of responses to the above messages (based on the previous discussion) are presented in Appendix A.

4. The **bfs Algorithm**

This algorithm deviates slightly from the **dfs algorithm**. The main points of difference are:

(1) Searches are sent, when appropriate, to all neighbors of a node. This implies that the search is performed in parallel and that more than one search with a certain round number may reach a graph node. Thus, when a bfs reaches a node it may be the first search with his round number to reach this node, or it may find that the node is already marked with its own round number, or it may find that there are other pending searches with the same round number at that node.

(2) Adding the arc \((k,l)\) at node \( k \) may initiate a search towards \( l \) if \( \text{tenr}(k) \geq \text{largest}[l] \). This search is necessary because bfs assumes that a search is going on along all branches from \( k \).
(3) The priority is now (an upper bound on) the number of nodes along the longest path in the graph starting at the node and traversing nodes whose round numbers are equal. Note that this definition will also allow satisfying the invariant INV.

Consider a SEARCH message \((j,r,p)\) arriving at node \(k\). If \(r<estr(k)\), the response \((k,r,0)\) will be returned, while if \(r=estr(k)\), an immediate response \((k,r,estp(k))\) is returned: Observe that in bfs the third component in the SEARCH message is extraneous and no useful information is propagated downwards through it. If \(estr(k)<r\leq tenr(k)\), the search waits as before, with the difference that several searches with the same round number but different sources may be present. Thus a set of sources with a given round number and priority (so far) will be waiting together. If \(r>tenr(k)\), then this search "takes over" and searches are initiated in parallel through \(k\)'s descendants, in addition to waiting for the result in Waiting\((k)\).

Consider an ANSWER message \((j,r,p)\) that updated \(\text{largest}[j]\), i.e. a search sent from \(k\) through \(j\) has returned and it is possible that many descendants of \(j\) are marked with round number \(r\). The third component, \(p\), of this message indicates the maximum possible number of nodes on any path starting at node \(j\) and traversing nodes whose round number equals \(r\). This number is a worst case upper bound as some of these nodes may already be marked with larger round numbers due to other ongoing searches in the graph. The argument \(p\) is compared to the known maximum path lengths for round number \(r\) (these may have been reported by previous ANSWER messages). If this search reports a longer possible path length, this new length is recorded in the triple \((ws,wr,wp)\) in Waiting\((k)\) for which \(wr=r\). Observe that if \(\text{largest}[j]\) is not updated by the ANSWER message, then no descendant of \(j\) is marked with round number \(r\) and hence the message update to \(p\) is irrelevant and concerns "ancient history" somewhere in the graph; it is therefore ignored.
If the ANSWER message \((j,r,p)\) updated \(\text{largest}[j]\) where \(j=\text{ind}(k)\), then \(\text{estr}(k)\), \(\text{ind}(k)\) and \(\text{estp}(k)\) are recomputed as before, and all searches with small enough round numbers are returned. The bfs algorithms are also described in Appendix A.

5. Correctness of the algorithm

Some of the informal claims made in the previous sections will now be precisely stated, and their proofs outlined. The theorems are proved for both the dfs and bfs methods in the absence of deletions and definitive searches.

Theorem 1. In both the dfs and bfs versions, for every pair of nodes \(i\) and \(j\) such that \(j \in \text{Neighbors}(i)\), the following invariant is true:

\[ \text{estr}(i) < \text{estr}(j) \lor (\text{estr}(i) = \text{estr}(j) \land \text{estp}(i) > \text{estp}(j)) \]

Proof: Note first that for all \(i\), \(\text{estr}(i)\) is monotonically increasing over time. The invariant was true when \(j\) was added to \(\text{Neighbors}(i)\), since it was checked directly (in the responses to RETURNVAL- or ANSWER). Increasing the round number marking of \(j\) preserves the invariant, and the priority of \(j\) is never increased for the same round number once it is first established. Thus the only difficulty could be in incorrectly re-marking node \(i\). Round numbers \(r\) sent in ANSWER messages are always less than or equal to the established round number of the source of that message, and \(\text{largest}[j]\) (which is also monotonically increasing) is equal to one of these \(r\) values from \(j\). Thus \(\text{estr}(i) \leq \text{largest}[j] \leq \text{estr}(j)\).

If \(\text{estr}(i) < \text{estr}(j)\), the invariant holds. Otherwise, the two are equal, which could only occur if, in effect, \(\text{ANSWER}(j,\text{estr}(j),\text{estp}(j))\) had earlier been sent from \(j\) to \(i\) as a response to a search. But upon receiving that message, \(wp\) is updated to be at least equal to the value of \(\text{estp}(j)\) in the triple \((s,\text{estr}(j),wp)\) in Waiting\((i)\). When \(\text{estr}(i)\) was set to the same value as \(\text{estr}(j)\) (either immediately or later on), the value of \(wp\) can only have possibly increased, and \(\text{estp}(i)\) will then be set to \(wp+1\), so that \(\text{estp}(j) \leq wp < \text{estp}(i)\). Thus the priority at \(i\) is larger than that at \(j\), as...
required, and the invariant always holds.

Theorem 2. For both versions, there are no cycles in the graph.

Proof: Assume there is a cycle. By Theorem 1, there can be no drop in the round numbers between nodes anywhere along the cycle. This is only possible if the round numbers are all equal. But in that case, by Theorem 1 all of the priority numbers must be decreasing between any two adjacent nodes on the cycle. Since this is impossible, there can not be a cycle in the graph.

Theorem 3. For each $SEARCH(j, r, p)$ message sent to $k$, there is a subsequent corresponding $ANSWER(k, r, p')$ message with the same $r$ sent to $j$ within finite time. That is, each re-marking will terminate.

Proof: First we show that the theorem holds if we assume that $k$ receives an $ANSWER$ message corresponding to every $SEARCH$ message which it sends to a neighbor. As soon as $SEARCH(j, r, p)$ is received at $k$, either the corresponding $ANSWER$ message is sent immediately if $estr(k) > r$, or $tenr(k)$ is or becomes at least as large as $r$, and a $SEARCH(k, r', p')$ is (or was previously) sent along the arc $ind(k)$, with $r' \geq r$. After the corresponding $ANSWER$ message is received, $largest[ind(k)]$ will be at least as large as $r$. Then either $estr(k) \geq r$ and the desired $ANSWER$ message will be sent from $k$ back to $j$, or there is another arc $(k, i)$ which has the smallest value of $largest[i]$, so that $ind(k)$ will be changed to $i$. A $SEARCH$ message with a round number equal to $tenr(k) \geq r$ will eventually be sent on $i$, either because a new $SEARCH$ increases $tenr(k)$, or because an $ANSWER$ message with a round number equal to $tenr(k)$ will be received at $k$. This process continues until the $largest[i]$ values are greater than or equal to $r$ for all neighbors $i$. Then $estr(k)$ will be at least $r$, and the desired $ANSWER$ message will be sent.

Note that new arcs can be added at $k$ while this process is going on. However, the number of arcs from an individual node is bounded by $N$. Hence searches cannot be indefinitely delayed due to new edge additions requiring new searches.
From the above argument, it follows that the only way in which a SEARCH which arrives at $k$ might not have a corresponding ANSWER from $k$ is if some SEARCH sent from $k$ to a neighbor does not have a corresponding ANSWER. However, the same reasoning can be applied to the neighbor node recursively. Since there is no cycle in the graph (by Theorem 2), there would have to be a SEARCH message sent to a leaf node (i.e., a node with no neighbors) for which there is no corresponding ANSWER message returned. However, it is obvious by inspection of the algorithm, that for any SEARCH message reaching a node with no neighbors, there is an immediate ANSWER message. The above argumentation could be formalized using temporal reasoning and proof lattices as in [7].

6. Deletions

Allowing deletions of arcs is, in principle, a trivial extension, since circuits cannot be closed by deletions. However, some implications for the inspection algorithm for addition of arcs must be considered. First, it now becomes possible to adopt a strategy of simply retrying the inspection algorithm even if a clear negative answer has previously been obtained for a proposed arc. Of course, the idea is that an arc which was in the graph might have meanwhile been deleted, making the desired addition possible.

Second, the values in the node involved must be updated. As for addition of arcs, a deletion of an arc $(k,j)$ must be directed to the node $k$. Clearly, $j$ should be removed from Neighbors($k$), but in addition largest[$j$] in $k$ now becomes undefined. As a result, if ind($k$)=j both estr($k$) and ind($k$) must be recomputed. If there should now be no neighbors, ind($k$) is set to 0 and estr($k$) should be set to tenr($k$). Otherwise estr($k$) will be the minimum of the remaining largest[$i$] values. This may, as when an ANSWER message was received, allow continuing some of the searches in Waiting($k$), by sending ANSWER messages back from $k$ when estr($k$) becomes as large as the waiting search round number.
Third, the possibility that a SEARCH message has been sent along an arc which is about to be deleted before the corresponding ANSWER has been received must be considered. As seen in Theorem 3, the correctness of the algorithm depends on the fact that a search is going on in the descendants of \( k \) which will eventually allow "releasing" all the searches pending at \( k \). For the bfs version a SEARCH message has already been sent along all arcs from \( k \), and at least one such search has not yet returned along all arcs \((k,i)\) with \( \text{largest}[i] < \text{tenr}(k) \). Thus no further steps are needed.

On the other hand, for the dfs version, a difficulty arises if the arc deleted was the one on which the last (and possibly only) SEARCH message was sent, and no corresponding ANSWER has yet been received. The straightforward solution is simply to wait until that search returns. Note that it will return, even though it is not doing any marking which will directly benefit the searches waiting at \( k \). The reader is invited to check that the response to the ANSWER message arriving from the node \( j \), after \((k,j)\) has been deleted, will automatically send a new SEARCH message along the arc designated by the new \( \text{ind}(k) \) (if it is still needed, i.e., \( \text{ind}(k) \) is not 0), even though \( j \not\in \text{Neighbors}(k) \). Therefore the argumentation of Theorem 3 still holds.

In this solution the priority of a waiting search may be unnecessarily increased, but this has no effect on the correctness of the algorithm. It is also possible to avoid the unnecessary delay by defining a new, "dummy" tentative round number, increasing \( \text{tenr}(k) \) and starting a search from \( k \) with that round number. Since this leads to several complications in the algorithms, we do not pursue this possibility here.

The code for treating the DELETE message may be found in Appendix B.
7. Definitive searches

7.1 Basic strategy

A definitive search DS is initiated when upon completion of a search it is still unclear whether the edge from i to j may be safely added; such a search is initiated only when it is judged vital to determine a conclusive search result. (Observe that a regular search may be tried any number of times prior to a decision to use a definitive search.)

At any point in time, at most one definitive search is allowed to operate in the graph. This restriction may be enforced by several methods. One possibility is to designate one node as 'central'. This node receives requests for definitive searches. It queues these requests, allowing only one to operate. Another possibility is to assign a 'weight' to each node. Based on weight, a 'leader' is chosen (e.g. by using the method in [3]). The leader is allowed to perform a definitive search. Following the search it lowers its weight to allow other interested nodes a chance to be elected as leaders.

The advantage in allowing only one DS at a time is in avoiding the necessity to handle "collisions" between DSs. Hence, the problem of integrating DS into our scheme separates into two parts where the first part, choosing a leader, does not affect normal graph operations. Assuming some mechanism exists for ensuring that at most one definitive search operates at any point in time, we now proceed to describe the details of a definitive search. Let node i be the node initiating a definitive search DS at j due to a request to add the edge from i to j to the graph.

To facilitate its integration into our scheme, DS should resemble an ordinary search as much as possible. For an ordinary search, it may repeatedly happen that even though the edge from i to j may safely be added, each time the search completes, the invariant condition does not hold between node i and node j. In a possible scenario, while the search s initiated by i at j is operating, estr(i) is incremented due to actions taken by searches originating elsewhere in the graph. As the rate of progress of nodes in performing operations may vary, whenever s
terminates it finds that \( estr(i) \) is "too big". This leads to the requirement that \( estr(i) \) should not be allowed to increase beyond the value of newr chosen as the round number for the definitive search, as long as that search is going on.

Not allowing \( estr(i) \) to increase unrestrictedly effectively "freezes" node \( i \) in a limited way. When \( estr(i) \) is to be incremented (due, say, to the return of some search to \( i \)), incrementing may have to be delayed until DS terminates. This may "block" the return of some searches from node \( i \). Note that DS itself may reach \( i \) and at some point its actions may require incrementing \( estr(i) \) to the newr chosen for this definitive search. However, incrementing up to newr is allowed, and does not cause blocking, so that deadlock is avoided.

If DS were to operate completely like an ordinary search then it might wait at some node for the return of a search which is blocked at node \( i \). Thus, in order to ensure that DS eventually terminates, DS never waits (like an ordinary search), instead it goes on "chasing" previous searches. The details are described in 7.2.

### 7.2 DS

Consider a definitive search DS initiated by node \( i \) at node \( j \) with round number newr. After establishing that node \( i \) has permission to initiate the next (unique) definitive search, node \( i \) freezes itself with respect to DS. That is, any update of \( estr(i) \) to a value larger than newr is not performed, and is queued. Of course, any waiting searches must continue to wait. ANSWER messages on behalf of DS or those with round numbers less than newr are treated normally: these messages can lead to updating \( estr(i) \), and cause the return of waiting searches as usual. Any other messages which are not ANSWER messages are unaffected.

Next, DS traverses the graph in a depth first search pattern equipped with newr as its round number. As in ordinary searches, DS "recoils" when it encounters a node \( k \) with \( estr(k) > newr \). DS differs from ordinary searches in that it ignores \( tenr(k) \). If \( newr > estr(k) \) then DS proceeds searching towards \( \text{ind}(k) \). When DS returns to node \( k \) via an ANSWER message, if after all updating the rela-
tion newr>estr(k) still holds, then DS continues searching towards the current
ind(k). So, DS never waits at a node for the return of some other searches with a
larger round number.

When DS terminates and returns to node i, if estr(i)<newr the edge addition
is performed. Otherwise, estr(i)=newr, and the addition is prohibited. Finally,
node i unfreezes itself and resumes normal operation (by first processing the
queued actions).

**Theorem 4:**

(1) DS preserves the invariant between nodes it visits.

(2) Upon DS's termination, a definite answer is reached regarding the edge addi-

tion.

(3) DS eventually terminates.

**Proof:**

(1) In this respect DS behaves as an ordinary search.

(2) Since node i is frozen while DS traverses the graph, the only way estr(i) can
reach the value newr is if the increase is caused by DS reaching node i. Hence, if
upon DS's final return to node i, estr(i)=newr then there was a way to reach node
i from node j and hence the edge addition request is denied. If
estr(i)<newr ≤ estr(j), the edge may clearly be added.

(3) The graph has n nodes and hence at most \(n^2\) SEARCH messages will be sent on
behalf of DS. As DS never waits for another search's completion, DS eventually
terminates. Note that deletions affect DS in the same way they affect ordinary
searches.

Observe that only in the absence of deletions does estr(i)=newr imply that
there is now a path from node j to node i. If deletions are present, the conclusion
from estr(i)=newr is that there *may be* a path from node j to node i as there *was*
a way to reach node i from node j. Finally, note that the eventual termination of
DS guarantees eventual unfreezing at i. This in turn ensures that our previous
proof concerning the termination of ordinary searches still holds.
8. Conclusions

The problem of edge addition/deletion under an acyclicity constraint has been examined. Two methods were outlined - using depth first and breadth first searches.

The approach used was the transformation of sequential algorithms for solving the above problem into distributed algorithms. The sequential algorithms maintain an invariant which holds after every operation; this invariant guarantees acyclicity. The main effort in designing the distributed version was invested in maintaining the invariant at all times. Otherwise, concurrent operations may misinterpret what is for one of them merely a "temporary inconsistency". One important benefit of the dependence on an invariant is that the correctness of the algorithm, as expressed in Theorems 1-3, is relatively easily provable.

The interactions among the concurrent activations of the operations at the various nodes lead to optimizations in which the task of one operation is partially performed by another. On the other hand, the degree of distributiveness leads to possible uncertainty for negative responses, which may be resolved by a restricted "freezing" mechanism. This mechanism, definitive search, is shown to be easily incorporated into the proposed schemes.

REFERENCES


Appendix A: The Distributed Algorithm

The variables at a node $k$ will be:

- $\text{estr}(k)$ -- the established round number at $k$
- $\text{estp}(k)$ -- the established priority number at $k$
- $\text{tenr}(k)$ -- the largest round number waiting at $k$
- $\text{Neighbors}(k)$ -- a set of the neighbors of $k$ in the (acyclic) graph
- $\text{largest}[i]$ for all $i \in \text{Neighbors}(k)$ -- the largest established round number passed back to $k$ from the subgraph with its origin at $i$
- $\text{ind}(k)$ -- a node identifier such that $\text{largest[\text{ind}(k)]=estr}(k)$
- $\text{Waiting}(k)$ -- a set of the searches waiting at $k$. A search is denoted by a triple $(\text{set_of_source_nodes}, \text{roundnum}, \text{prioritynum})$.
- $\text{Wantoadd}(k)$ -- a set of nodes at which $k$ has initiated a remarking and for which no decision has been reached.

Both the dfs and bfs versions of the algorithm are described. For the most part, the versions are identical. Differences are indicated via a case statement, "case search_kind of", differentiating between the two search versions.

Initially, $\text{Neighbors}(k)$, $\text{Waiting}(k)$, and $\text{Wantoadd}(k)$ are empty, while $\text{estr}(k)$ and $\text{tenr}(k)$ are zero, as are the variables $\text{ind}(k)$ and $\text{largest}[i]$. The variable $\text{estp}(k)$ is initially set to $k$. The algorithm for a node $k$ will be described in terms of the messages and the corresponding responses to them:

For $\text{ADDARC}(j: \text{nodename})$ do /*a request to add $(k,j)$ */
send $\text{GETVAL}(k)$ to $j$; /*gather $j$'s marking */

For $\text{GETVAL}(j: \text{nodename})$ do /*a request from $j$ for $k$'s marking */
send $\text{RETURNVAL}(k, \text{estr}(k), \text{estp}(k))$ to $j$.
For RETURNVAL(j:node, r:roundnum, p:prioritynum) do
/* a response to a previous request, giving j's marking */
if estr(k) < r or (estr(k) = r and estp(k) > p)
/* first check of the invariant */
then begin
  if empty(Neighbors(k)) then ind(k) := j; /* initialize ind(k) */
  add j to Neighbors(k); /* add the arc (k, j) */
  largest[j] := estr(k); /* initialization */
  case search_kind of
    bfs: if tenr(k) > estr(k) then
      send SEARCH(k, tenr(k), 0) to j;
    dfs: /* do nothing */
    endcase
end
else begin /* the initial check fails */
  choose unique newr > tenr(k);
  /* newr > estr(k) is sufficient, but unwise */
  add j to Wantoadd(k);
  send SEARCH(k, newr, 0) to j /* start a search from */
  end;
end;

For SEARCH((j:node, r:roundnum, p:prioritynum) do
/* a request from J for a remarking */
if r < estr(k)
then send ANSWER(k, r, p) to j /* no need to continue */
else if r = estr(k)
/* again, no need to continue */
then case search_kind of
  dfs: send ANSWER(k, r, p) to j
  bfs: send ANSWER(k, r, estp(k), to j
  endcase
else /* r > estr(k) */
if empty(Neighbors(k)) then /* can return, after updating */
begin estr(k) := r;
  estp(k) := p + 1;
  tenr(k) := r;
  send ANSWER(k, r, p + 1) to j
end
else begin /* there are neighbors of k to be remarked */
  case search_kind of
    dfs: add ([j], r, p) to Waiting(k);
    bfs: if there is a (ws, wr, wp) E Waiting(k) with r = wr
      then add j to ws
      else add ([j], r, p) to Waiting(k)
  endcase
  if r > tenr(k) then /* this search should continue */
  begin tenr(k) := r;
    case search_kind of
      dfs: /* continue at smallest descendant */
        send SEARCH(k, r, p) to ind(k);
      bfs: /* search all descendants */
        send SEARCH(k, r, p) to all i E Neighbors(k)
    endcase
  end
end;
For ANSWER(j:nodename; r:roundnum; p:prioritynum) do

/* j has completed remarking */
if j ∈ Wantoadd(k)
    then begin /* want to add (k,j), after a search */
        if estr(k)<r or estr(k)=r and estp(k)>p
            then begin /* the invariant now holds between k and j */
                if empty(Neighbors(k)) then ind(k) := j; /* initialize */
                add j to Neighbors(k);
                largest[j] := estr(k); /* estr(k) is OK as is */
                remove j from Wantoadd(k);
                case search_kind of
                    bfs: if tenr(k)>r
                        then send SEARCH(k, tenr(k), 0) to j;
                        else largest[j] := tenr(k)
                    endcase
            end
        else if estr(k)=r then /* cant add */
            begin remove j from Wantoadd(k);
                announce failure to add to initiator
            end
        else /* when estr(k)>r, don't know */
            may initiate a definitive search
    end
else if estr(k)<r then /* otherwise message should be ignored */
    begin
        largest[j] := r;
        if there is (ws,wr,wp) ∈ Waiting(k) with wr=r then
            if wp < p then wp := p; /* for dfs the test is true */
        if j=ind(k) then
            begin /* estr(k) and ind(k) must be recomputed */
                estr(k) := min{ largest[i] | i ∈ Neighbors(k) };
                ind(k) := i ∈ Neighbors(k) such that largest[i]=estr(k);
                /* any such i is acceptable */
                if there is (ws,wr,wp) ∈ Waiting(k) with wr=estr(k) then
                    begin wp := wp+1, estp(k) := wp end;
                for all (ws,wr,wp) ∈ Waiting(k) and wr<=estr(k) do
                    begin /* return all completed searches */
                        remove (ws,wr,wp) from Waiting(k);
                        for all s ∈ ws send ANSWER(k, wr, wp) to s
                    end
                end
            endcase
        dfs:
        if estr(k)<r and r=tehr(k) then send SEARCH(k, r, p) to ind(k);
        bfs: /* searches were originally sent to all neighbors */
    endcase
end
Appendix B: Deletion

For DELETE(j:nodename) do
/* a request to delete (k,j) */
if j ∈ Neighbors(k) then
begin
  remove j from Neighbors(k);
if empty(Neighbors(k)) then
  begin
    ind(k):= 0;
estr(k):= tenr(k)+1;
estp(k):= 1
  end
else
  if j=ind(k) /* must update established roundnumber */
  then begin
    estr(k):= min{larget[i] | i ∈ Neighbors(k)}
    ind(k):= i ∈ Neighbors(k) with largest[i]=estr(k);
    for the (ws,wr,wp) ∈ Waiting(k) with wr=estr(k) do
    begin
      wp:=wp+1,
estp(k):=wp
    end;
    for all (ws,wr,wp) ∈ Waiting(k) with wr<=estr(k)
    begin remove (ws,wr,wp) from Waiting(k);
      for all s ∈ ws do
        send ANSWER(k,wr,wp) to s
    end
  end
else if j ∈ Wantoadd(k) then remove j from Wantoadd(k)