COOPERATIVE DISTRIBUTED ALGORITHMS FOR DYNAMIC CYCLE PREVENTION (Revised Version)

by

S. Katz and O. Shmueli

Technical Report #380
October 1985
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 edges in a directed graph without creating a cycle. Such algorithms are useful for a variety of problems in distributed systems, such as preventing deadlock or ordering priorities. The algorithms operate in a realistic asynchronous computer network environment in which there are numerous possible interactions among overlapping instances of the algorithms.

The distributed algorithms are derived from a sequential algorithm. In developing distributed versions 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

In this paper, we treat the problem of adding and deleting edges in a distributed directed graph without creating a cycle. The model of computation considered assumes asynchronous distributed processes, with no shared memory. All communication is by message passing, and all messages are assumed to arrive in finite time, although no FIFO or other ordering is postulated. Full communication possibilities are assumed, i.e., any process can communicate with any other. To complete the list of assumptions, processes are assumed to have unique identifiers, and the total number of processes is known to be bound (although the exact number is not available to the processes).

Each process will be associated with a node of the graph to be stored, and will maintain a list of the nodes to which an edge is directed from that node. The solution to the problem must be a prevention algorithm since a request to add an edge can be granted only if no cycle will thereby be created in the directed graph being maintained. As in other prevention algorithms, a trivial solution can be obtained by always refusing to add the requested edge. Clearly, a desirable property of a solution can be informally identified: a request will be denied only when absolutely necessary. Moreover, since the model is distributed, a solution should exploit this fact and not involve what is commonly known as freezing. More precise explanations of these terms will be given later in the paper.

The motivation for this problem and model comes from a number of system tasks such as distributed deadlock prevention and distributed concurrency control [2,5]. However, it can also be viewed as an example of the more general task of maintaining a distributed data structure modified by parallel, overlapping operations. Note that the graph does not represent the communication possibilities between nodes, since nodes not connected in the graph will need to exchange messages about their states. It can, however, be interpreted as distributing other global information such as priorities between processes, or the fact that a process must wait for a resource held by another process.

In order to obtain some feeling for the difficulty of the problem, consider a "solution" in which a request to add an edge from node i to node j is treated by searching for a path from j to i. If such a path is found, clearly the requested edge cannot be added. However, if no other steps are taken, then even if no such path is found from j to i, process i cannot be sure that a path does not exist. An edge from
node $k'$ to node $l$, with $k$ reachable from $j$ and $l$ reachable from $i$, could have been added to the graph since $k$ was checked, but before the edge from $i$ to $j$ was added. Thus a positive response to the question of whether the edge can be added is unreliable if this solution is used. Any straightforward attempt to overcome this difficulty will involve preventing all descendants of $j$ in the graph from adding edges for some time, which is an unacceptable freezing of much of the graph. Another instance of the same basic difficulty is that it is necessary to prevent the simultaneous attachment of two or more edges which together close a cycle.

The methodology used in developing the solution to be given here is felt to be applicable to other problems in maintaining distributed data structures with overlapping operations. In the first stage, a global, sequential algorithm is found to solve the problem. This solution is based on an invariant which must be true before and after each (sequential) abstract operation, but not necessarily while the implementation of the operation is executing. In the next stage, the algorithm is distributed among the processes, and the programs at each node are adjusted so that the invariant holds at all times, including during the implementations of individual operations. This is necessary so that operations which execute in parallel will not be "misled" by temporary inconsistencies. During this stage, 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.

As the first step in our proposed methodology, we describe a global, sequential solution where no parallelism is involved. Such solutions have been proposed in [6] and [8]. The sequential algorithm is based on the fact that a directed graph $G$ is acyclic if there exists a ranking function $r$ from the vertices to a set of values with a strict order relation $<$ such that if there is an edge from $i$ to $j$ then $r(i) < r(j)$. The ranking function will be represented by a variable rank associated with each node $i$. A graph $G$ with such variables is conservative if whenever there is an edge from a node $i$ to a node $j$ in $G$ then the condition $(\text{rank}_i < \text{rank}_j)$ holds. This condition will be called $\text{INV}(i,j)$. A conservative graph is clearly acyclic.
Suppose the edge from i to j is to be added to a conservative graph G. If INV(i,j) holds between i and j, then the edge 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 reranking may be attempted in order to satisfy INV(i,j) by increasing j's rank, and thus allow the edge's addition. In the sequential versions two methods may be used for reranking, based on depth first search (dfs) and breadth first search (bfs) [1]. Below and in most of this paper, the depth-first version is treated. Some of the considerations needed for a breadth-first methodology are examined in Section 6.

In the depth-first sequential method, a new value larger than all the existing rankings (including i's) is chosen, and a depth first reranking from j is started. In this reranking, j will have a new rank of the chosen value, and each node reachable from j must be given a new rank larger than that of its predecessors. Since there is no cycle, such a reranking can always be successfully completed. When the reranking is completed, INV(i,j) is checked again. Now, there is no path from j to i iff INV(i,j) is true: since the algorithm is sequential, the only way that i could now have a rank larger than j's new ranking is if there were a path from j to i. In Figure 1, the steps for adding an arc in this environment are outlined in pseudo-code.

if \( \text{rank}_i < \text{rank}_j \) then add the arc from i to j
else begin
    choose a value v larger than all existing rank values;
    \( \text{rank}_j := v \);
    Conduct a depth-first traversal of the descendants of j increasing the rankings encountered as needed so that afterwards, INV is true between all connected nodes;
    if \( \text{rank}_i < \text{rank}_j \) then add the arc from i to j
    else refuse to add
end;

Figure 1: Global sequential algorithm for edge addition
2. Developing a Distributed (depth-first) Algorithm

In the distributed environment, the assumptions of the model are crucial in developing and understanding the algorithms. As noted previously, here graph nodes are considered as processes communicating by message passing. Each process has a unique identifier, local memory and a fair (not necessarily FIFO) message queue. Node $i$ will contain the information that it is connected by edges 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 edge, and the graph should be considered as a data structure maintained distributively by the processes. It is, however, true that while reranking nodes, messages usually (but not always) will be sent between nodes connected via an edge in the graph.

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

One approach to incorporating a cycle-prevention algorithm into a message-passing environment is to designate a single central control process, which receives all requests to add an edge. Even without considering possible algorithms, such a central node solution has serious drawbacks. The central node may become a communication bottleneck due to the large volume of messages. The entire system would be paralyzed by a failure of that process. Requests which in fact are completely unrelated may be scheduled in the artificial order of arrival at the central node. These are all symptoms of freezing, which is the arbitrary delay of the implementation of an operation for reasons which are unrelated to the requirements of that operation. Fortunately, these difficulties can be avoided, by using the sequential algorithm as a basis for developing a totally distributed solution. Because the solution is distributed, there may be several rerankings going on at the same time. As will be further explained later, it may then happen that no reranking will end up with a conclusive answer. This is likely only under high request load. Should a conclusive answer be essential, in Section 5 a mechanism called definitive reranking is described which will refuse to add an edge only when such a refusal is unavoidable. The possible cost of such a reranking will be a limited freezing of some other rerankings.
Given the global sequential solution, and the nature of the distributed environment, some of the properties of the distributed algorithm are immediate. First, a request to add an edge from i to j should be directed to i, where the checks of the invariant should be performed, and the actual addition of the edge is done (should this be safe?). This is necessary so that there is no danger of rank increasing between the check and the actual addition of the edge, possibly leading to a non-conservative graph.

In the distributed version, no process has global knowledge of the current largest rank value, and at any rate new rankings definitely can begin elsewhere in the graph while previous ones are still going on. Thus the value of a reranking can not be guaranteed to be the largest in the graph. However, note that when i is initiating a reranking at j due to a request to add the edge from i to j, it is sufficient to choose any value larger than ranki. In order to guarantee that the rank values chosen for different rerankings be distinct, each rank can be written as a rational number whose integer component is chosen to be sufficiently large and whose fractional (less significant) component is i’s (unique) node identifier. Thus by choosing distinct integer components locally, each node can generate globally unique numbers.

Continuing in the ‘translation’ of the sequential algorithm for edge addition, the initial check of the relation between the ranks of nodes i and j cannot be done directly, since i has no knowledge of j’s rank. Thus an exchange of messages is needed, in which i requests j’s rank and, at some later time, j sends back a message with its rank. If the value sent from j is larger than i’s rank at the moment of the inspection by i, the edge can be safely added by i. This is true because i will not change its own rank until the edge is added, and even if j has in the meantime changed its rank, the value could have only increased. Hence the graph will remain conservative when the edge is added. The fact that rankings increase monotonically is therefore crucial to the correctness. One way to overcome the potential unboundedness of the rankings is outlined in Section 7.

When the initial check is unsuccessful, a reranking is attempted. Since numerous requests will be processed in parallel, the graph must be conservative not only before or after the rerankings, but also during their execution. The depth first reranking will be initiated by a message from i to j, and clearly messages must be sent in turn along all edges emanating from j in the graph, ‘activating’ the reranking.
there. In order to keep the graph conservative, the actual changes in the rank of a node must be done from the leaves back towards the process (node) where the reranking originated. Otherwise there would be a moment at which the rank of the source of an edge would be larger than that of its target. Thus the message activating the reranking will be passed along by a node \( n \), but only when its descendants in the graph have been appropriately reranked, can the rank of \( n \) itself be changed.

In addition, a simple use of integer or rational numbers as rank values, with the usual 'less than' relation, is problematic in this context. This is because when a node is reranked, the longest path from the original initiator (\( j \), above) is not known, and it is not clear by how much to increase its rank. This could lead to a node being given a series of new rank values in the course of a single reranking operation, which is clearly undesirable. In order to avoid this problem, a slightly more complex ordering relation will be used, over rankings which are pairs of values. The first component will be called the round and in a reranking will be chosen by \( i \) to be a unique value larger than \( i \)'s own present round value. Unlike the previous rank, the round will remain constant and be assigned to all nodes reranked by that reranking operation. The second value will be called the priority and will be incremented from the leaves during a reranking with a given round number. The priority will be increased in the order in which the nodes are last visited in the depth-first traversal. A rank value thus becomes a pair \((\text{round}, \text{priority})\) with the relation \( \text{rank}_m < \text{rank}_a \) iff

\[
\text{round}_m < \text{round}_a, \text{or } \left( \text{round}_m = \text{round}_a \right) \text{ and } \left( \text{priority}_m > \text{priority}_a \right).
\]

Note that if two such pairs of values have the same round numbers, then the pair with the larger priority will be "smaller" in this ordering. As is proven in Section 3, this relation defines an ordering which is also sufficient to prevent a cycle, if the graph is always conservative, using this definition of the ordering. This relation will also be denoted as \( \text{INV}(j, j) \). It remains to decide what should be maintained at each node in order to conduct such rerankings, and what should occur when such a reranking request reaches a node \( k \). The node \( k \) clearly must maintain its round and priority numbers (referred to in the sequel as the established values and denoted by \( \text{est}(k) \) and \( \text{estp}(k) \), respectively); as well as information about rerankings which are going on through \( k \), but have not yet been completed.

A node will receive a structured message activating a reranking with a proposed round number \( r \), a priority \( p \), and the immediate-source of the message, \( s \). As an initial intuition, \( p \) can be seen as the
number of nodes given the round number \( r \) so far. Due to various later modifications, this intuition will ultimately not be accurate, but the real requirement of keeping the graph conservative still will be maintained whenever \( p \) is used to assign a new established priority at a node. Several situations are possible and must be treated:

(1) a reranking message first reaches node \( k \) and the established round number, \( \text{estr}(k) \), is larger than or equal to the round number of the desired reranking, \( r \). This means that in any continuation of the reranking from \( k \), all the nodes encountered have been marked with round numbers larger or equal to \( \text{estr}(k) \) (and thus to \( r \)). It is forbidden to change the round numbers of those nodes to \( r \), since then the graph might cease to be conservative (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 reranking from \( k \) with round \( r \). The reranking should simply "retreat" to node \( s \) (i.e., a response message should be sent back to \( s \)) and continue the reranking 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 \( \text{INV}(i,j) \) will not hold when the reranking terminates. If the round number is equal to \( r \), the priority to be returned will be larger than the priority of \( k \), so that the graph remains conservative.

(2) a reranking reaches \( k \) with \( \text{estr}(k) < r \). This does not necessarily mean that the new reranking can immediately continue, since another reranking, 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 \( \text{tenr}(k) \) (tentative round number) which is the value of the largest round number in a reranking ongoing through \( k \).

Two situations must be considered:

(2a) \( \text{estr}(k) < r \) and \( r \leq \text{tenr}(k) \). By reasoning similar to case (1), continuing the reranking with \( r \) is a waste of effort, since all of the descendants of \( k \) will ultimately be marked with a larger or equal round number. However, in this situation, the reranking 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 \( \text{tenr}(k) \). Thus the new reranking should simply wait until it can safely retreat. (It should be noted that, since there are no cycles in the graph, \( r = \text{tenr}(k) \) can only occur when deletions are allowed, as discussed in Section 4.)
(25) \( estra(k) < r \) and \( r > tenr(k) \). Now the reranking \((s, r, p)\) should "take-over" the task of increasing the ranks of the descendants of \( k \). The variable \( tenr(k) \) is to be updated to the value of \( r \), and the reranking should be continued at one of the neighbors of \( k \).

For each edge from \( k \) to \( l \), we now consider the maximum of the round numbers from rerankings which have \( returned \) along that edge 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 \( estr(k) \) will 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 reranking returns along an edge \( l \) with a round number \( r \), the round numbers of all nodes reachable through \( l \) are at least as large as \( r \). Thus the largest value of the round for that edge may be updated; possibly leading to a recomputation of \( estr(k) \) and the priority. Then all round numbers of rerankings waiting at \( k \), including \( r \), should be compared with \( estr(k) \). All waiting rerankings with round numbers less than or equal to \( estr(k) \) may resume operation by sending an answer message to the immediate source of their reranking message, and thus cease waiting at \( k \). Note that if a larger round number than \( r \) had previously been returned along \( l \), then the new message with \( r \) is irrelevant and can be ignored.

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

Consider a reranking resulting from a request for adding the edge from node \( i \) to node \( j \). When the reranking is completed, and a message is returned from \( j \) to the initiating node \( i \), \( \text{INV}(i,j) \) is checked. If it holds, then the edge is added. If it's round number equals the reranking round number, then there was a way to reach node \( i \) from node \( j \), and the edge addition request is denied. Otherwise, the reranking terminates \( incondisively \). As already indicated, the ways in which this may occur and how to nevertheless obtain a definitive response to the request are described in Section 5.
Each node must potentially fulfill a role (1) as the receiver of a request to add an edge, (2) as the origin of a reranking (which is the target of the edge to be added), and (3) as a node encountered during a reranking. The messages received at a node \( k \) are:

- **ADDARC** \((j)\) -- an external request to \( k \) (possibly generated by other tasks at \( k \) ) to add the edge \((k, j)\)
- **GETVAL** \((j)\) -- a request from \( j \) to \( k \) for the established values of round and priority
- **RETURNVAL** \((j, r, p)\) -- a response from \( j \) to a previous **GETVAL** \((k)\) request
- **RERANK** \((j, r, p)\) -- a request from \( j \) for a reranking with round number \( r \) and a priority so far of \( p \)
- **ANSWER** \((j, r, p)\) -- a response to a previous **RERANK** 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 variables at a node \( k \) are:

- \( \text{estr}(k) \) -- the established round number at \( k \)
- \( \text{estp}(k) \) -- the established priority number at \( k \)
- \( \text{tenr}(k) \) -- the largest round number from a triple in \( \text{Waiting}(k) \), defined below
- **Neighbors** \((k)\) -- the set of neighbors of \( k \) in the (acyclic) graph
- \( \text{largest}[i] \) for all \( i \in \text{Neighbors}(k) \) -- the largest established round number returned from \( i \) in **ANSWER**
- **ind** \((k)\) -- a node identifier such that \( \text{largest}[\text{ind}(k)] = \text{estr}(k) \)
- **Waiting** \((k)\) -- the set of rerankings at \( k \) for which a **RERANK** message has been received, but no corresponding **ANSWER** message has been sent from \( k \) in response. Rerankings with the same round number will be grouped together into a single triple (set-of-sources, roundnum, prioritynum) so that they are differentiated only by having been sent from different immediate source nodes.
- **Wantoadd** \((k)\) -- the set of nodes at which \( k \) has initiated a currently active reranking.

Initially, **Neighbors** \((k)\), **Waiting** \((k)\), and **Wantoadd** \((k)\) are empty, while \( \text{estp}(k) \) and \( \text{tenr}(k) \) are zero, as are the variables **ind** \((k)\) and **largest**[\(i\)]. The variable **estr** \((k)\) is initially set to a unique function of \( k \), for example, "0.k". The messages and corresponding responses described below are to be executed sequentially within a node \( k \), in response to any fair order of incoming messages.

For **ADDARC** \((j, \text{nodename})\) do

```
  /* a request to add \((k, j)\) */
  send **GETVAL** \((k)\) to \( j \) /* gather j's rank */
```
For GETVAL(j:node name) do /* a request from j for k's rank */
    send RETURNVAL(k:estr(k),estp(k)) to j;

For RETURNVAL(j:node name;r:round num;p:priority num) do
    /* a response to a previous request, giving j's rank */
    if estr(k) < r or (estr(k) == r and estp(k) > p)
        /* first check of the invariant */
        the begin
            if empty(Neighbors(k)) then ind(k) := j; /* initialize ind(k) */
                add j to Neighbors(k); /* add the edge (k,j) */
                largest[j] := estr(k); /* initialization */
            end
        else begin /* the initial check fails */
            choose unique newr > tenr[k];
                /* newr > estr(k) is sufficient, but unwise */
                add j to Want to add(k);
                send RERANK(k,newr,0) to j /* start a reranking from j */
            end;

For RERANK(j:node name;r:round num;p:priority num) do
    /* a request from j for a reranking */
    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 begin
            if estp(k) >= p then p := estp(k) + 1;
                send ANSWER(k,r,p) to j
            end;
        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 reranked */
            if there is a (i,j,wr,wp) in Waiting(k) with r = wr
                then begin add j to ws;
                    if p > wp then wp := p;
                    end;
            else add (i,j,r,p) to Waiting(k);
                if r > tenr[k] then /* this reranking should continue */
                    tenr[k] := r;
                /* continue at smallest descendant */
                    send RERANK(k,r,p) to ind(k);
                end;
        end;

    /* continue at smallest descendant */
    send RERANK(k,r,p) to ind(k);
For ANSWER(j:nodeName; r:roundnum; p:priorityNum) do

  /* has completed reranking */
  if j ∈ WantoAdd(k)
    then begin
      /* want to add (k, j), after a reranking */
      if estr(k) < r or estr(k) == r and estp(k) > p
        then begin
          /* INV(k, j): now holds between k and j */
          if empty(Neighbors(k)) then ind(k) := i /
              /* initialise */
          add i to Neighbors(k);
          largest[i] := estr(k) /
              /* estr(k) is OK as is */
          remove j from WantoAdd(k);
        end
      else if estr(k) == r then /* cannot add */
        begin remove j from WantoAdd(k);
          announce failure to add to initiator
        end
      else /* when estr(k) > r, don’t know! */
        may initialise a definitive reranking
    end
  else /* the continuation of a regular reranking */
    if r > largest[j] then /* otherwise message should be ignored */
      begin
        largest[j] := r;
        if there is a (ws, wr, wp) ∈ Waiting(k) with wr ≤ r and wp < p
          then wp := p;
        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 for some i ∈ Neighbors(k) such that largest[i] = estr(k);
            /* any such i is acceptable */
            if there is a (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) with wr ≤ estr(k) do
              begin
                /* return all completed rerankings */
                remove (ws, wr, wp) from Waiting(k);
                for all s ∈ ws do
                  send ANSWER(k, wr, wp) to s end;
            end
          end
        if estr(k) < r and r = tenr(k) then send RERANK(k, r, p) to ind(k);
      end
    end;

3. 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 proven for the depth-first traversal method in the absence of deletions and definitive rerankings.
Theorem 1: For every pair of nodes $i$ and $j$ such that $j \in Neighbors(i)$, the following is invariantly true:

$$estr(i) < estr(j) \lor (estr(i) = estr(j) \land estrp(i) > estrp(j))$$

Proof: Note first that for all $i$, $estr(i)$ is monotonically increasing over time and, once it is assigned, the value of $estrp(i)$ is not changed for a given round number. The assertion $estr(i) < r \lor (estr(i) = r \land estrp(i) > p)$ was true when $j$ was added to $Neighbors(i)$, since it was checked directly (in the response to RETURNVAL or ANSWER). Since $r \leq estr(j)$ and if $r = estr(j)$ then $p = estrp(j)$, the assertion in the theorem is true when the edge is added. Increasing the round number of $j$ preserves the invariant. Thus the only difficulty could be in incorrectly reranking node $i$. From the definitions of the variables it is easy to establish that $estr(i) \leq largest(j) \leq estr(j)$.

If $estr(i) < estr(j)$, the invariant holds. Otherwise, the two are equal, which could only occur if, in the worst case, ANSWER($j,r,p$) had earlier been received at $i$ from $j$, with $r = estr(j)$ and $p = estrp(j)$, and $i$ has not changed its ranking since sending that message. But upon receiving that message, $wp$ was updated to be at least equal to the value of $estrp(j)$ in the triple $(wsp,rwp)$ in $Waiting(i)$. When $estr(i)$ was set to the same value as $estr(j)$ (either immediately or later on), the value of $wp$ can only have possibly increased, and $estrp(i)$ will then be set to $wp + 1$, so that $estrp(i) \leq wp < estr(i)$. Thus the priority at $i$ is larger than that at $j$, as required, and the invariant always holds.

Theorem 2. There are never any 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 cannot be a cycle in the graph.

In order to prove that each reranking will eventually terminate, the following definition is useful: a RERANK($j,r,p$) message from $j$ to $k$ is answered if there is a subsequent ANSWER($k,r,p'$) message sent from $k$ to $j$, having the same $r$ value.

Lemma 1. For any computation of the network, if a RERANK($j,r,p$) message sent to $k$ is never answered, then there must be another message RERANK($k,r',p'$) sent by $k$ to one of its neighbors.
which is also never answered in the computation.

Proof: Suppose that there is a computation such that a RERANK \((j,r,p)\) message to \(k\) is never answered, but each RERANK message sent from \(k\) is answered. As soon as RERANK \((j,r,p)\) is received at \(k\), either the corresponding ANSWER message is sent immediately if \(\text{estr}(k) > r\), or \(\text{tenr}(k)\) is or becomes at least as large as \(r\), and RERANK \((k',r',p')\) is (or was previously) sent along the edge \(\text{ind}(k)\), with \(r' \geq r\). By the above supposition, the corresponding ANSWER message will be received, and then \(\text{largest}[\text{ind}(k)]\) will be at least as large as \(r\). Then either \(\text{estr}(k) \geq r\) and the desired ANSWER message will be sent from \(k\) back to \(j\), or there is another edge \((k,i)\) which has the smallest value of \(\text{largest}[i]\), so that \(\text{ind}(k)\) will be changed to \(i\). A RERANK message with a round number equal to \(\text{tenr}(k) \geq r\) will eventually be sent on \(i\), either because a new RERANK message increases \(\text{tenr}(k)\), or because an ANSWER message with a round number equal to \(\text{tenr}(k)\) is received at \(k\) (either the message just received, or a later one). Since \(k\) has only finitely many neighbors, eventually the largest \([i]\) values will be greater than or equal to \(r\) for all neighbors \(i\). Then \(\text{estr}(k)\) will be at least \(r\), and the desired ANSWER message will be sent back to \(j\). This contradicts the supposition that the RERANK \((j,r,p)\) message is never answered, thereby proving the Lemma.

Note that new edges can be added at \(k\) while this process is going on. However, the number of edges from an individual node is bounded by \(N\). Hence rerankings cannot be indefinitely delayed due to new edge additions requiring new rerankings.

Theorem 3. Each RERANK message is answered. That is, for each RERANK \((j,r,p)\) message sent to \(k\), there is a subsequent ANSWER \((k,r,p')\) message with the same \(r\) sent to \(j\) within finite time.

Proof: by contradiction. Consider a RERANK \((j,r,p)\) message to \(k\) which is never answered. By Lemma 1, there must be an infinite sequence \(k_1, k_2, \ldots\) of nodes such that \(k_i\) and \(k_{i+1}\) are neighbors and \(k_i\) sent a RERANK \((k_i,r_i,p_i)\) message to \(k_{i+1}\) which is never answered. Since there are finitely many nodes, some \(k_i\) must appear more than once in this sequence. However, this could only occur if there were a cycle in the graph, which is not true from Theorem 2. Thus there is a contradiction.

The proof above could be formalized using temporal reasoning and proof lattices as in [7].
4. Deletions

Allowing deletions of edges is, in principle, a trivial extension, since circuits cannot be closed by deletions. However, some implications for the inspection algorithm for addition of edges must be considered. For example, 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 edge. Of course, the idea is that an edge which was in the graph might have meanwhile been deleted, making the desired addition possible.

More importantly, the values in the node involved must be updated. As for addition of edges, a deletion of an edge \((k,j)\) must be directed to the node \(k\). Clearly, \(j\) should be removed from \(\text{Neighbors}(k)\), but in addition \(\text{largest}[j]\) in \(k\) now becomes irrelevant to rankings waiting at \(k\). As a result, if \(\text{ind}(k) = j\) both \(\text{estr}(k)\) and \(\text{ind}(k)\) must be recomputed. If there should now be no neighbors, \(\text{ind}(k)\) is set to \(0\) and \(\text{estr}(k)\) should be set to \(\text{tenr}(k)\). Otherwise \(\text{estr}(k)\) will be set to the minimum of the remaining \(\text{largest}[i]\) values. This may, as when an ANSWER message is received, allow continuing some of the rerankings in \(\text{Waiting}(k)\), by sending ANSWER messages back from \(k\) when \(\text{estr}(k)\) becomes as large as the waiting reranking round number.

As seen in Theorem 3, the correctness of the algorithm depends on the fact that a reranking is going on in the descendants of \(k\) which will eventually allow releasing all the rerankings waiting at \(k\). Thus, a difficulty arises if the edge deleted was the one on which the last (and possibly only) RERANK message was sent, and no corresponding ANSWER has yet been received. The straightforward solution adopted here is simply to wait until the RERANK message is answered. Note that it will be answered, even though the reranking in former descendants will not directly benefit the rerankings waiting at \(k\).

The response to the ANSWER message arriving from the node \(j\), after \((k,j)\) has been deleted, will automatically send a new RERANK message along the edge designated by the new \(\text{ind}(k)\) (if it is still needed, i.e., \(\text{ind}(k)\) is not \(0\)), even though \(j \notin \text{Neighbors}(k)\).

An additional implication of deletions is that a released reranking may reach a node, along an alternative path, which has been or is being reranked with the same round number due to leftover messages in descendants of \(i\). As always, we need only ensure that the graph remains conservative. In
particular, if a request \((s, r, p)\) reaches a node \(d\) with \(estr(d) = r\), the priority values must be considered. If the established priority at \(d\), \(estp(d)\), is smaller than the priority so far in the request message, \(p\), then there is no problem in the rankings, and the reranking can be continued at \(s\). Otherwise, \(p\) should be set to \(estp(d) + 1\), and then continue reranking at \(s\). This is already incorporated into the algorithm for RERANK. There may also now be several rerankings waiting at a node with the same round number, something which could not have occurred without deletions. However, nowhere do we depend on the fact that only one request for a reranking with a given round number is active at any moment (we require only that one source originally created the round number and initiated a reranking). In this solution the priority of a waiting reranking may be unnecessarily increased, but this has no effect on the correctness of the algorithm.

The code for treating the DELETE message is given below:

For DELETE(j
odename) do
  /* a request to delete \((k, j)\) */
  if \(j \in \text{Neighbors}(k)\) then
    begin
      remove \(j\) from \(\text{Neighbors}(k)\);
      if empty(\(\text{Neighbors}(k)\)) then
        begin
          ind\((k)\) := 0;
          estr\((k)\) := tehr\((k)\) + 1;
          estp\((k)\) := 1
        end
      else if \(j = \text{ind}(k)\) /* must update established round number */ then
        begin
          estr\((k)\) := \(\min\{\text{largest}[i] \mid i \in \text{Neighbors}(k)\}\);
          ind\((k)\) := \(\text{for some } i \in \text{Neighbors}(k) \text{ with largest}[i] = estr(k)\);
          if there is a \((ws, wr, wp) \in \text{Waiting}(k)\) with \(wr < estr(k)\) do
            begin \(wp := wp + 1\);
              estp\((k)\) := wp
            end
          for all \((ws, wr, wp) \in \text{Waiting}(k)\) with \(wr \leq estr(k)\) do
            begin remove \((ws, wr, wp)\) from \(\text{Waiting}(k)\);
              for all \(a \in w s\) do
                send ANSWER\((k, wr, wp)\) to \(s\)
            end
        end
      else if \(j \in \text{Wantoadd}(k)\) then remove \(j\) from \(\text{Wantoadd}(k)\)
  end

It is clear that the graph remains conservative after a deletion. We also have:
Theorem 4. Even when deletions are allowed, each RERANK message is answered.

Proof: Lemma 1 still is true when deletions are allowed, and if a RERANK message is not answered, there must still be an infinite sequence of nodes as in the proof of Theorem 3. However, due to deletions, a node can appear more than once in such a sequence. If node $k$ appears infinitely often in the sequence, its round number must have increased (by at least 1) between every two RERANK messages it sent which were not answered, i.e., between every two appearances in the sequence. But then the first requested reranking waiting at $k$ will eventually be answered, so that no such infinite sequence can exist.

5. Definitive rerankings

A definitive reranking DR is initiated when upon completion of a reranking it is still unclear whether the edge from $i$ to $j$ may be safely added; such a reranking is initiated only when it is judged vital to determine a conclusive reranking result. (Observe that a regular reranking may be tried any number of times prior to a decision to use a definitive reranking.) Such a situation occurred because the rank at $i$ was increased beyond the new chosen when the regular reranking was initiated at $j$. Thus an uncertainty reminiscent of that for the incorrect solution mentioned in the introduction has appeared. The crucial difference is that there, the uncertainty is whether a positive response still holds, while here the less crucial question is whether a refusal is justified. In the former case, ignoring the problem would lead to closing cycles, while here it would merely lead to an occasional unjustified refusal.

One way to solve the problem is to conduct a special reranking while freezing all other activity in the network. This solution is clearly undesirable since it prevents completely unrelated rerankings from progressing. Fortunately, such a radical solution is also unnecessary. However, in order to avoid deadlock situations, it is necessary to require that each node may initiate only one definitive reranking at a time, so there will be no more than $N$ active at any one time.

To facilitate its integration into our scheme, DR should resemble an ordinary reranking as much as possible. For an ordinary reranking, it may repeatedly happen that even though the edge from $i$ to $j$...
may safely be added, each time the reranking completes, the invariant condition does not hold between node $i$ and node $j$. In a possible scenario, while the reranking initiated by $i$ at $j$ is operating, $estr(i)$ is incremented due to actions taken by rerankings originating elsewhere in the graph. As the rate of progress of nodes in performing operations may vary, whenever $s$ terminates it finds that $estr(j)$ 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 reranking, as long as that reranking is active.

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 reranking to $i$), incrementing may have to be delayed until DR terminates. This may block the return of some rerankings from node $i$. Note that DR itself may reach $i$ and at some point its actions may require incrementing $estr(i)$ to the newr chosen for this definitive reranking. However, incrementing up to newr is allowed, and does not cause blocking, so that deadlock is avoided.

If DR were to operate completely like an ordinary reranking then, it might wait at some node $k$ for the return of a reranking with a larger round number but which is frozen at node $i$. Thus, in order to ensure that each DR eventually terminates, a DR never waits (like an ordinary reranking), instead it goes on "chasing" previous rerankings. However, it may itself be temporarily frozen at a node which has initiated another definitive reranking (with a lower round number).

Consider a definitive reranking DR to be initiated by node $i$ at node $j$ with round number newr. After establishing that node $i$ has no other ongoing definitive reranking at that moment, node $i$ freezes itself with respect to the new DR. That is, any update of $estr(i)$ to a value larger than newr is not performed, and is queued. Of course, any waiting rerankings must continue to wait. ANSWER messages on behalf of DR 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 rerankings as usual. In general, any messages to $i$ which do not increase $estr(i)$ beyond newr are unaffected.

Next, DR traverses the graph in a depth first reranking pattern from $i$, equipped with newr as its round number. As in ordinary rerankings, DR "recoils" when it encounters a node $k$ with $estr(k) > newr$. DR differs from ordinary rerankings in that it ignores $tenk(k)$. If $newr > estr(k)$ then
DR proceeds reranking towards $\text{ind}(k)$. When DR returns to node $k$ via an ANSWER message, if after all updating, the relation $\text{newr} \geq \text{estr}(k)$ still holds, then DR continues reranking towards the current $\text{ind}(k)$. So, DR never waits at a node for the return of some other reranking with a larger round number, but it can be frozen due to another DR.

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

Theorem 5:

1. The graph remains conservative when a DR is executed.
2. Upon DR's termination, a definite answer is reached regarding the edge addition.
3. Each DR eventually terminates.

Proof:

1. In this respect DR is identical to an ordinary reranking.
2. Since node $i$ is frozen while DR traverses the graph, the only way estr(i) can reach the value newr is if the increase is caused by a definitive reranking reaching node $i$. Hence, if upon DR'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. A definitive reranking does not wait for other rerankings, but could be frozen at a node $k$ which initiated a definitive reranking with a smaller round number. However, since at most one definitive reranking is allowed at one time for each node, and since the number of nodes is bounded, there can be at most $N$ definitive rerankings at a time, and the smallest one must terminate. If new round numbers increase locally by at least 1, each frozen definitive search will either terminate or eventually be the smallest, and then terminate. Note that deletions affect DR in the same way they affect ordinary rerankings.

Observe that only in the absence of deletions does estr($j$) = 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 $j$. Since there is no way short
of yet again checking the descendants of j to establish whether a deletion has taken place, there is no choice but to refuse the requested addition of the edge \((i,j)\). Finally, note that the eventual termination of each DR guarantees eventual unfreezing at i. This, in turn ensures that our previous proof concerning the termination of ordinary rerankings still holds.

6. A breadth-first version

This algorithm differs only slightly from the depth-first algorithm. The main differences are:

1. Rerankings are sent, when appropriate, to all neighbors of a node. This implies that the reranking is performed in parallel and that more than one reranking with a certain round number may reach a graph node. When a breadth-first reranking reaches a node, it may be the first with its 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 rerankings with the same round number at that node.

2. Adding the edge \((k,i)\) at node k may initiate a reranking towards l if \(tenr(k) > \text{largest}(l)\). This reranking is necessary because a breadth-first strategy assumes that a reranking is going on along all branches from k.

3. The priority is now the largest number of nodes traversed in returning from a leaf node which was given the round number of the reranking. It will be (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 and guarantee acyclicity.

Consider a RERANK message from j with proposed round number r, arriving at node k. If \(r < \text{estr}(k)\), the response ANSWER\((k,r,0)\) will be returned, while if \(r = \text{estr}(k)\), an immediate ANSWER\((k,r,\text{estp}(k))\) is returned. In this version the third component in the RERANK message is extraneous and no useful information is propagated downwards through it. Thus it will be omitted in the breadth-first algorithm. If \(\text{estr}(k) \leq r \leq \text{tenr}(k)\), the reranking waits as before, except that several rerankings with the same round number but different sources may be simultaneously present, even without deletions. If \(r > \text{tenr}(k)\), then this reranking takes over, and rerankings 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 reranking 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 rerankings 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 reranking 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.

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 rerankings with small enough round numbers are returned. The breadth-first algorithms are described in Appendix-A.

The worst case complexities of the breadth-first and depth-first approaches in a centralized sequential environment are identical. However, assuming all edge addition requests are equally likely, the depth-first 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 reranking. A similar estimate for the breadth-first method is unknown.

7. Bounding the range of round numbers

The requirement that round numbers must only increase seems to imply an unbounded range of values, which is, at least theoretically, disturbing. One obvious solution to this difficulty is to globally shut down the system from time to time, disallowing all new messages and resetting all of the rankings with small values so that the graph will again be conservative when normal network operation is resumed. Such a solution may be the easiest practical approach, but it is clearly global and may interfere with normal operation. In order to incorporate boundedness without global freezing, we will use modular arithmetic and only increase rankings. The key idea is to divide the available range of values for
round numbers into three regions to be called for convenience 0, 1, and 2. Each process knows the boundaries of the regions, and thus knows which region its round number is in. At any moment, at most two of the regions can be in use (i.e., nodes may have round numbers only from those regions). The remaining region will be called vacant.

In order for the scheme described below to be reasonable, each region should be significantly larger than N, the bound on the number of nodes. It is also important that initially all nodes have round numbers from the same region, say region 0. This already follows from the initialization.

In the remainder of this section, addition modulo 3 will be used when referring to regions (so \(2+1=0\)). The new ordering used in comparing round numbers is \(a > b\) iff, for some i

\((a, b \in \text{region}(i)\) and \(a > b\) or \((a \in \text{region}(i+1)\) and \(b \in \text{region}(i)\))

Since it will be invariantly true that at least one region is vacant, the above ordering will not be cyclic. In order to maintain this invariant property, a node (process) \(m\) in region \(i\) may rerank itself to a value which is in region \(i+1\) only when region \(i+2\) is vacant. In order to determine whether this is already the case, \(m\) has three local boolean variables \(\text{vacant}_n(k)\) for \(k=0, 1, 2\), to denote whether region \(k\) is vacant. For any \(j\) and \(k\), \(\text{vacant}_n(k) = \text{true}\) if \(k\) is vacant. Note that the other direction does not hold, so that \(k\) could be vacant even if \(\text{vacant}_n(k) = \text{false}\). Initially, for each \(m\), \(\text{vacant}_n(2) = \text{true}\), and the other boolean variables are false.

If \(\text{vacant}_n(i+2)\) is true locally, the regions in use are \(i\) and \(i+1\), and the node can safely rerank into \(i+1\). Otherwise, the node \(m\) which would like to rerank into \(i+1\) must initiate evacuation of region \(i+2\). This will be done by increasing all values in \(i+2\) into region \(i\), which is obviously not vacant since \(m\) is in that region.

In order to evacuate region \(i+2\), \(m\) will broadcast to every other node a \(\text{vacate}(i+2)\) message. When a node \(j\) receives such a message, \(\text{vacant}_n(i)\) must already be false (since \(j\) is clearly in \(i\)). The node \(j\) will check what region its round number is in. If it is already not in region \(i+2\), a \(\text{vacated}(i+2)\) message will be returned to \(m\). Otherwise, an "internal request" is generated to rerank \(j\) with a (low) round number in region \(i\). This request will be treated completely normally, according to the algorithms presented. As soon as \(j\) is reranked into \(i\), either due to the internally generated request, or as part of
any other reranking, j will return to m a vacated(i+2) message. Note that if indeed j was in region i+2, then region i+1 must actually be vacant, since both i+2 and i are being used.

When such messages have been received at m from all nodes, then m will broadcast an allvacated(i+2) message and wait for an acknowledgement from all nodes. When this is received, m can increase its round number into region i+1. A node m receiving an allvacated(i+2) message will set vacant(i+2) to true, set vacant(i+1) to false, and return an acknowledgement message. Note that if several nodes would like to enter region i+1, the first to succeed has already guaranteed that vacati(i+2) is true in every process before it actually enters i+1, and the others can use their local variable to see that it is safe.

In order to incorporate this scheme into the existing algorithms, whenever new - (a perspective round number value) is to be generated, the conditions for crossing a region boundary must be checked first, and regular operation suspended locally when an evacuation is required. The global knowledge needed in order to use the technique suggested here is the boundaries of the regions, and the actual number of existing nodes in the graph, so that a node can validate that a response has been received from all nodes. In fact it is sufficient that indirect communication eventually take place, as long as the initiating node is able to determine when all other nodes have responded. This would allow optimizations such as having an underlying spanning tree among the nodes in order to handle broadcast and acknowledgement.

8. Conclusions

The problem of edge addition/deletion under an acyclicity constraint has been examined. Once the basic depth-first algorithm was described and proven correct, variations and additions to cover deletions, definitive response, breadth-first reranking, and bounding the range of rank values were presented.

Much recent work on deadlock (e.g., 3) has advocated detection and breaking of cycles after they occur, rather than prevention. However, in the more general dynamic context considered here, it is not clear that detection is either possible or desirable. Only when all needed safety properties of a
system are maintained even after the creation of a cycle, is detection feasible. Note that it is likely that when a detection strategy is used, the algorithm would "successfully" treat a graph which always has cycles. It is not even clear that detection is more efficient than prevention. In some cases, the detection algorithm may have to be executed after each addition of an edge. Recall that in the prevention algorithm given here, in about half of (randomly likely) requests to add an edge, no reranking at all would be needed, and INV will be true for the initial inspection.

The approach advocated was the transformation of a sequential algorithm for solving the problem into distributed algorithms. The sequential algorithm maintains an invariant which holds after every operation; this invariant guarantees cyclicity. 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 algorithms, as expressed in Theorems 1-4, is proven relatively easily.

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 reranking, was shown to be easily incorporated into the proposed schemes.

REFERENCES


Appendix A: The Breadth-first Distributed Algorithm

The variables and initial values are the same as for the depth-first version. The responses to the messages ADDARC and GETVAL are unchanged, and are not repeated here.

For RETURNVAL( j:nodeName; r:roundnum; p:prioritynum ) do
/*a response to a previous request, giving j's rank */
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) */
  . . .
end;
else begin /*the initial check fails */
  if tenr(k) > estr(k) then begin
    send RERANK(k, tenr(k)) to i;
  end;
end;

For RERANK( j:nodeName; r:roundnum ) do /* a request from j for a ranking */
if r < estr(k)
then send ANSWER(k,r,0) to j /*no need to continue*/
else if r = estr(k) then /*again, no need to continue*/
  then send ANSWER(k,r,estp(k)) to j;
else /*r > estr(k)*/
  if empty(Neighbors(k)) then /*scan, return after updating*/
    begin estr(k) := r;
      . . .
    end;
else begin /*there are neighbors of k to be ranked*/
  if there is a (wr, wr, w) ∈ Waiting( k ) with r = wr
    then add j to wr.
  else add (j, r, 0) to Waiting( k );
  if r > tenr(k) then /*this reranking should continue*/
  begin tenr(k) := r;
  end;
  send RERANK(k,r) to all i ∈ Neighbors(k)
end;
end;
For ANSWER(i;hostname; r;roundnum; p;prioritynum) do
  /* j has completed reranking */
  if j ∈ Wantoadd[k]
    then begin /* want to add (k,j) after a reranking */
      if estr(k) < r or estr(k) = r and estp(k) > p
        then begin /* INV(k,j) 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);
          if tenr(k) > r
            then send RERANK(k,tenr(k)) to j;
          else largest[j] := tenr(k)
        end
      else if estr(k) = r then /* can't 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 reranking */
    end
  else /* the continuation of a regular reranking */
    if r > largest[j] then /* other message should be ignored */
      begin
        largest[j] := r;
        if there is (ws,wr,wp) ∈ Waiting(k) with wr = r and wp < p
          then wp := p;
        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 */
          end
        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 rerankings */
            remove (ws,wr,wp) from Waiting(k);
            for all s ∈ ws send ANSWER(k,wr,wp) to s
          end
      end
  end;