Concise Papers

An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights

K. B. LAKSHMANAN, K. THULASIRAMAN, AND M. A. COMEAU

Abstract—In this paper, we are interested in the design of distributed algorithms for the single-source shortest-path problem to run on an asynchronous directed network in which some of the edges may be associated with negative weights, and thus a cycle of negative total weight may also exist. The only one existing solution in the literature for this problem is due to Chandy and Misra, and it has, in the worst case, an unbounded message complexity. We first describe and study a synchronous version of the Chandy–Misra algorithm and prove that for a network with $m$ edges and $n$ nodes the worst case message and time complexities of this algorithm are $O(mn)$ and $O(n)$, respectively. This algorithm is then combined with an efficient synchronizer to yield an asynchronous protocol that retains the same message and time complexities.

Index Terms—Analysis of algorithms, asynchronous network, distributed computing, message and time complexities, negative cycle, protocols, shortest paths, synchronizer.

I. INTRODUCTION

Let $G = (V, E)$ be a directed network in which $V$ is the set of nodes and $E$ is the set of edges, with $|V| = n$ and $|E| = m$. Each edge $(i, j)$ has an associated weight $w(i, j)$. The classical single-source shortest-path problem in such a network is to find the shortest paths from one distinguished node $s$, called the source, to every other node. We assume that every node is reachable from $s$. Note that each node is reachable from itself. The distance $d(i)$ of a node $i$ is the weight of the shortest path from the source node to the node $i$. If the weights associated with some of the edges are negative, then a cycle of negative total weight may also exist. In this case, if a negative cycle is reachable from $s$, then all nodes on this cycle are said to be at a distance $\propto$ from the source. Moreover, every node reachable from a negative cycle is also said to be at a distance $\propto$ from the source. For all remaining nodes, the distance values are finite, and hence their shortest paths are well defined. Networks with negative weights arise in the context of several applications [4], [7], [12], [17].

In this paper, we are interested in distributed algorithms for the single-source shortest-path problem in a directed network in which a cycle of negative total weight may exist. At the end of the computation, the algorithm should identify nodes which are at a distance $\propto$ from the source, besides computing the shortest paths for the nodes at a finite distance. We assume the existence of a processor at each of the nodes and that these processors are interconnected through communication links as indicated by the edges of the network. The exchange of messages between two processors is asynchronous in that the sender always hands over the message to the communication subsystem and proceeds with its own local task. The communication subsystem, we assume, will deliver the message to its destination, without loss or any alteration, after a finite but undetermined time lapse. The messages sent over any link also follow a first-in, first-out rule. A detailed description of this model of distributed computing may be found in [5], [14], [16]. In our distributed algorithm, all messages are required to be of fixed length, specifying the type of the message and some Boolean parameters, except for a few types of messages that may carry distance estimates, i.e., the weight of some path from the source node $s$. The message complexity of the distributed algorithm is the total number of messages transmitted during the execution of the algorithm. The time complexity is the time that elapses from the beginning to the termination of the algorithm, assuming that the propagation delay in any link is at most one unit of time.

A synchronous network differs from an asynchronous one in the existence of a global clock, so that all network messages are sent only when a clock pulse is generated. More importantly, a message sent by any node to its neighbor arrives at its destination before the end of the clock period, i.e., just before the next pulse [1], [2], [13]. A synchronizer is a mechanism that helps simulate a synchronous network on an asynchronous one. Recently, Awerbuch [1] has proposed the use of synchronizers as a general methodology for designing efficient distributed algorithms for asynchronous networks.

For the case of networks with nonnegative edge weights, a number of distributed algorithms for the single-source shortest-path problem have been proposed in the past [6], [8], [10], [11]. The only existing solution for the problem of finding distributively the shortest paths in an asynchronous network with negative cycles is due to Chandy and Misra [5]. They construct an algorithm which can be viewed as a distributed implementation of the sequential Ford–Bellman–Moore algorithm, combined with an elegant way to detect negative cycles through additional messages. The mechanism Chandy and Misra use to detect negative cycles is based on the earlier work of Dijkstra and Sholten [9] in the context of termination detection for diffusing computations. The emphasis in their paper is on correctness arguments rather than on a complexity analysis. It can be shown that, if there is a negative cycle, the number of messages exchanged in the Chandy–Misra algorithm cannot be bounded. Even if there is no negative cycle, the pattern of delays in communication links could be such that the number of messages exchanged becomes exponential in the number of nodes [3].

In this paper, we describe an asynchronous distributed algorithm whose worst case message and time complexities are $O(mn)$ and $O(n)$, respectively. We permit negative cycles in the network. Our approach is to obtain a synchronous version of the Chandy–Misra algorithm and then combine it with an efficient synchronizer. An interesting aspect of the synchronizer we design is that messages can be handled one by one as soon as they arrive, in the same order, whether they are "too early" or are "out of date." Thus, the use of this synchronizer permits the execution of the Chandy–Misra algorithm to remain "totally asynchronous," as termed by Bertsekas and Eckstein [4].
II. CHANDY–MISRA ALGORITHM

In this section, we take a brief look at the asynchronous Chandy–Misra algorithm. The definitions of terms such as predecessor, successor, neighbor, etc., and a detailed proof of correctness as well as termination of the algorithm may be found in [5].

The Chandy–Misra algorithm basically works in two phases. In the first phase, all nodes at a finite distance from the source compute their final distance values, and in the second phase, nodes at distance = ∞ from the source are so informed. Two kinds of messages, LENGTH and ACK, are made use of in the first phase. Whenever a node discovers a path from s which is shorter than that currently known, it updates its distance and sends appropriate LENGTH messages to its successor nodes so that they can update their distances, if necessary. A node acknowledges each LENGTH message it receives by sending an ACK message. A LENGTH message is acknowledged by a node either 1) when this LENGTH message causes no update of the distance values of the node, 2) when the node has received ACK messages for all the LENGTH messages it transmitted to its successors, or 3) when a later LENGTH message arriving at the node causes an update of its distance value.

In the absence of a negative cycle, all nodes are at a finite distance from the source, and so the activity with LENGTH messages will eventually cease. The source node terminates execution of the first phase when all the LENGTH messages it sent out have been acknowledged. It is interesting to note that the same signaling scheme using ACK messages can be used to terminate execution of phase 1, even in the presence of negative cycles. Since nodes in or reachable from a negative cycle will continue to receive smaller and smaller LENGTH messages from other nodes, they will eventually release ACK messages for all the LENGTH messages received from nodes that are at a finite distance from the source. Thus, if the source node itself is not in a negative cycle, it will eventually receive all ACK messages for the LENGTH messages it sent out, at which point it will detect termination of the first phase of computation. If the source node is in a negative cycle, it terminates phase 1 when it receives a LENGTH message with a negative distance estimate.

Since the delays in the communication links are arbitrary, the ACK messages to be received by the source node, before it can terminate phase 1, could suffer long delays. In the meantime, the nodes in the negative cycles could keep sending messages in a cyclic fashion, updating their distances all the time. As a result, the worst case number of messages exchanged in the first phase of the asynchronous Chandy–Misra algorithm is really unbounded.

As regards phase II, the source node initiates this phase and broadcasts messages to inform all the nodes about the completion of phase I. When a node receives this message, it checks to see if ACK messages are yet to be received for some of the LENGTH messages it sent out. If so, the node recognizes the presence of a negative cycle and suitably communicates this information to all the other nodes. Clearly, the second phase requires only O(m) messages and O(n) time. Hence, to improve the performance, we need to concentrate only on phase I of the Chandy–Misra algorithm.

III. SYNCHRONOUS VERSION OF THE FIRST PHASE

The synchronous version of the first phase of the Chandy–Misra algorithm also requires two kinds of messages: LENGTH and ACK. The detailed set of actions to be initiated on receipt of these messages can be seen in the program segments of the Appendix. As in the asynchronous version, LENGTH messages are used to update distance estimates of successor nodes, whereas ACK messages are used to inform predecessor nodes that the corresponding LENGTH messages have been handled suitably. Since, in a synchronous protocol, messages can be transmitted only when a clock pulse is generated, the processing of messages does not trigger immediate dispatch of further messages. But suitable data structures are updated, and information regarding messages to be transmitted at the next clock pulse is recorded. For example, in the synchronous algorithm, if a LENGTH message arrives at the node i and causes an update of its distance estimate, a Boolean variable called change(i) is set to true, so that in the next pulse, the effect of the updated distance can be propagated to the successor nodes through appropriate LENGTH messages. Also, a data structure called ackset(i) is used to keep track of the identities of the predecessors in the previously known shortest paths for which ACK messages have to be sent. A careful analysis will show that, for the correct working of the algorithm, this data structure must really be implemented as a multiset, permitting as many as two copies of each element. When a new clock pulse is generated, the procedure newpulse is executed. At that time, all the LENGTH and ACK messages are transmitted. If the variable change(i) is true, it is switched back to false after sending LENGTH messages to successor nodes, until, of course, another LENGTH message causing an update of the distance estimate is received. The number of LENGTH messages propagated by the node i that are yet to be acknowledged is recorded in a variable called num(i). If we ignore ACK messages, it is clear that this synchronous algorithm is essentially a distributed implementation of the Ford–Bellman–Moore algorithm, with each pulse causing one sweep of all the edges [10].

The proof of correctness and termination of the above synchronous protocol follows along the same lines as in [5]. Consider num(i) and change(i), two pieces of data maintained by node i. As in the original asynchronous Chandy–Misra algorithm, at any time, node i will have at most one LENGTH message for which it has not generated an ACK message. This ACK message, if it is pending, should really go to the predecessor node in the current shortest path. Also, the fact that an ACK message is pending at node i is indicated by the truth of the compound Boolean condition (num(i) > 0 or change(i) = true). Thus, at the termination of phase I, this compound Boolean condition can be used to identify nodes in a negative cycle and those that are not. The following theorem summarizes these results.

Theorem 1: The synchronous version of phase I of the Chandy–Misra algorithm terminates, and at the end of that clock period and beyond, the following hold.

1) For each node i at a finite distance from the source s, d(i) contains its final stable distance value, and num(i) = 0 and change(i) = false.

2) For each node i at a distance = ∞ from the source, and only for such nodes, either num(i) > 0 or change(i) = true or there is a node j on a path from the source to i such that num(j) > 0 or change(j) = true.

As regards the complexity of the synchronous algorithm, we have the following result.

Theorem 2: The synchronous version of phase I of the Chandy–Misra algorithm terminates in O(n) clock pulses using O(mn) messages.

Proof: First, consider only those nodes at a finite distance from the source node. Since in a synchronous network all messages suffer at most one unit time delay, it follows that if the shortest path from node s to a node v consists of exactly one edge, then the distance value of node v will reach its final stable value at the end of the first clock period, i.e., just before the second clock pulse. Using this result, along with induction on the number of edges in a shortest path from s, it is easy to see that each node at a finite distance from the source will reach its final stable distance value before the nth pulse. Even after reaching the final distance value, each node will transmit at its next pulse LENGTH messages carrying this value, but will transmit no LENGTH messages thereafter because no further update of its distance is possible. Thus, after n pulses, only ACK messages can flow among the finite-distance nodes of the network.

Consider next a node v in a negative cycle having, say, e edges. Let the weight of this cycle, be –W where W > 0. Suppose at pulse k, node v receives a LENGTH message carrying a weight
equal to \( i \). This message will be acknowledged at the next pulse if it does not result in an update of the distance value of \( v \); otherwise, it will be acknowledged within the \((e + 1)\) pulses because during this period it will definitely receive a LENGTH message carrying a weight value \( c < W \). Thus, every LENGTH message received by node \( v \) will be acknowledged within \((e + 1)\) pulses of receipt of this message. In particular, any LENGTH message received at pulse height \( k \) from a finite distance node, say \( u \), will be acknowledged by node \( v \) at the \((k + e + 1)\)th pulse or earlier. Since \( k \) represents the number of edges in a path from the source node to the node \( v \), \((k + e) \leq n\). This means that after \((n + 1)\) pulses there will be no ACK messages flowing from nodes in negative cycles to finite-distance nodes.

Since ACK messages get propagated towards the source by triggering the dispatch of other ACK messages, it should be clear from the above that the source node, if it is not in a negative cycle, will receive ACK messages for all the messages it sent out within \(2n\) pulses and thus can detect termination of phase 1 in \( O(n)\) pulses.

If the source node is itself in a negative cycle, then it will receive a LENGTH message carrying a negative weight within the first \(n\) pulses and thus will recognize the presence of a negative cycle in \( O(n)\) pulses. We therefore conclude that the synchronous version of phase 1 will terminate in \( O(n)\) clock pulses.

Since at most one LENGTH message to any successor and two ACK messages to any predecessor are sent by a node during each clock pulse, the algorithm requires only \( O(mn)\) messages.

From the arguments presented in the above proof, it is clear that every node at a distance \(-\infty\) from the source will receive at least one LENGTH message after \(n\), but within \(2n\), clock pulses of its operation. Since it is possible to traverse the network, count the number of nodes, and deliver this information to each site before the shortest-path computation begins, using only \(O(m)\) messages and \(O(n)\) time, one can visualize an alternative algorithm in which each node terminates the execution after exactly \(2n\) clock pulses.

In this case, ACK messages are not needed, and each node can decide on its own whether or not it is at a distance \(-\infty\) from the source. However, our interest in this correspondence is to retain the elegance of the Chandy–Misra algorithm and modify it only to improve its efficiency. Hence, we use ACK messages. Moreover, if the shortest paths constructed contain very few edges compared to \( n \), use of ACK messages can indeed speed up the termination.

### IV. IMPLEMENTATION OF A SYNCHRONIZER

The synchronous algorithm presented in the previous section can be used on an asynchronous network, provided there is a way to simulate a sequence of clock pulses. A synchronizer is such a mechanism. In [15], we have drawn attention to certain difficulties one may face in the implementation of a synchronizer if we insist that the processing of a message cannot be delayed once received [16]. However, for the algorithm at hand, a simple and time-efficient synchronizer can be designed as discussed below. In contrast to the \(\alpha\)-synchronizer described in [1], our synchronizer requires fewer additional message types and also does not require the messages to carry the pulse number. More importantly, our synchronizer permits handling of messages one by one without additional delay, in the same order as they arrive, whether they are "too early" or "out of date," as explained later.

The basic idea in the synchronizer mechanism is as follows. The computation proceeds in "rounds," trying to simulate the actions of the synchronous algorithm by pulse. A node may start the actions corresponding to a new pulse once it is sure that the messages sent by its neighbors, if any, in the previous pulse of the synchronous algorithm have all arrived. For this purpose, each node waits for an explicit GO message from every one of its neighbors. Also, every node sending a message to its neighbor ensures that all useful messages of the synchronous algorithm are sent before a GO message. Thus, by the first-in, first-out property of the communication link, arrival of GO messages from all neighbors signals a node to initiate actions corresponding to the next pulse. Observe that GO messages basically add \(O(m)\) message and \(O(1)\) time complexity overheads per pulse of the synchronous algorithm.

In order to implement such a simulation, we need an initialization phase so that all nodes wake up and send GO messages to their neighbors. Then every node performs round one, corresponding to the first pulse of the synchronous algorithm. The program segments in the Appendix corresponding to WAKEUP and GO messages present all the actions to be taken in detail. The propagation of the WAKEUP message follows the protocol given by Segall [16]. Its message and time complexities are \(O(m)\) and \(O(n)\), respectively.

We remarked earlier that there are certain difficulties in the implementation and use of a synchronizer. This basically stems from the asynchronous nature of the network. Consider a data structure called gorceived \((i)\) needed at node \(i\) to keep track, at various stages, of the set of neighbors from which GO messages have been received. It is tempting to think that this set can be implemented at each node as a bit vector of size equal to the number of neighbors of that node. Unfortunately, this will not work. Consider a stage of execution when two neighboring nodes \(i\) and \(j\) have completed \((k - 1)\) pulses of the synchronizer. Then, each one must have transmitted a GO message, permitting the other to go through the \(k\)th round. But it is quite conceivable that before node \(i\) receives GO messages from all its neighbors and proceeds with the \(k\)th pulse, node \(j\) may receive all its GO messages and hence proceed to complete the actions corresponding to the \(k\)th pulse, thereby releasing another GO message to node \(i\). Thus, sometimes more than one GO message could arrive at node \(i\) from \(j\), and hence the data structure gorceived \((i)\) should really be implemented as a multiset, permitting more than one copy of an element. It is clear, however, that node \(j\) cannot start the \((k + 1)\)th pulse before node \(i\) completes its \(k\)th pulse and sends a GO message. Thus, at any point in time, the number of pulses of activity gone through by two neighboring nodes can differ by at most one. This has two implications. The first is that at any point in time there could be at most two GO messages received at a node from any of its neighbors. The second implication is that over the entire network, the number of pulses of activity gone through by any two arbitrary nodes can differ by as much as the length of the shortest path between these two nodes in the undirected communication network.

Consider now the execution of the synchronous protocol for phase I combined with the synchronizer proposed above. Since each node executes its \(k\)th pulse only after it is aware that each of its neighbors has executed the \((k - 1)\)th pulse and that no messages destined for it are still in transit, one may assume that execution of the combined protocols will pretty much be the same as that of the synchronous one. But there is a minor problem. We have already seen that if \(i\) and \(j\) are two neighboring nodes, there is the possibility that node \(j\) can complete the actions of its \(k\)th pulse after \((k - 1)\) pulses, but before it begins its \(k\)th pulse. This then implies the possibility of a message sent by node \(j\) in the \(k\)th pulse modifying some data structures maintained by node \(i\), in the process altering what node \(i\) would have done otherwise during the \(k\)th pulse of the synchronous algorithm. Suppose, for example, the distance estimate maintained at node \(i\) had changed as a result of a LENGTH message sent by a neighboring node during its \((k - 1)\)th pulse of activity. Normally, node \(i\) would propagate this fact to its successors through appropriate LENGTH messages only during its \(k\)th pulse. But suppose a neighboring node \(j\) completes its \(k\)th pulse before node \(i\) begins this pulse, and it sends a LENGTH message that causes a further update of the distance estimate maintained at node \(i\); then, the LENGTH messages sent by node \(i\) during its \(k\)th pulse will carry updated values. In other words, the LENGTH messages node \(i\) would have sent normally during its \(k\)th pulse do not get sent now. Moreover, the arrival of a LENGTH message from node \(j\) to node \(i\) can sometimes cause some other LENGTH message arriving at node \(i\) corresponding to the \((k - 1)\)th pulse of activity to be "out of date."
node are nonincreasing, and the values contained in the LENGTH messages flowing along any edge are always monotonically decreasing. Thus, "early" or "out-of-date" arrival of LENGTH messages at node \( i \) from neighboring nodes cannot cause an error in the computation. To be more precise, one can assert that the distance estimate maintained at node \( i \) before it executes its \( k \)-th pulse in the protocol simulated using the synchronized protocol will always be less than or equal to the distance estimate maintained before the execution of the \( k \)-th pulse in the synchronous version of the algorithm. Similarly, it is easy to observe that early arrival of ACK messages cannot cause an error in the computation. However, the data structure ackset(\( i \)), which keeps track of the predecessors for whom ACK messages have to be sent from node \( i \), will now have to be implemented as a multiset permitting as many as three copies of an element, whereas for the synchronous algorithm, an implementation as a multiset permitting at most two copies of an element would be sufficient. Such subtle issues make the implementation and use of synchronizers not a routine matter, but a crucial step requiring careful analysis and verification. As far as the sending and processing of ACK messages is concerned, it is possible to add a 2-bit parameter (to indicate a number between 1 to 3), so that multiple ACK messages to be sent one after another can be replaced by a single ACK message with the parameter suitably set.

In phase II, TERMINATE messages are propagated in a manner similar to that of the WAKEUP messages, but also using the results of Theorem 1 to identify nodes at a finite distance from the source node and those that are not. This phase is basically the same as in the original Chandy–Misra algorithm [5].

As regards the time and message complexities of phase I resulting from LENGTH and ACK messages, the results of Theorem 2 still hold because the source can detect termination of phase I before it executes no more than \( 2n \) pulses. Since at every other node the activity with go messages stops only after the first TERMINATE message arrives at that node, it could simulate by that time as many as \( 3n \) clock pulses. Since the simulation adds \( O(m) \) message and \( O(1) \) time overheads per pulse, the additional costs in message and time complexities are \( O(mn) \) and \( O(n) \), respectively. Thus, the overall complexity of the asynchronous algorithm resulting from LENGTH, ACK, WAKEUP, GO, and TERMINATE messages is only \( O(mn) \) in messages and \( O(n) \) in time. Clearly, the algorithm is time-optimal asymptotically.

V. Concluding Remarks

In this paper we have considered the single-source shortest-path problem in an asynchronous network with negative weights and modified the existing asynchronous Chandy–Misra algorithm so that the worst case message and time complexities are \( O(mn) \) and \( O(n) \), respectively. The algorithm is time-optimal asymptotically. In contrast, the original algorithm, the only one available so far in the literature for this problem, requires an unbounded number of messages in the worst case. The approach we have taken is to obtain a synchronous version of the Chandy–Misra algorithm and use a synchronizer to simulate the actions of the synchronous protocol on an asynchronous network. For this purpose, we have designed a simple and time-efficient synchronizer whose message and time complexity overheads are \( O(m) \) and \( O(1) \), respectively, per pulse of the synchronous algorithm. This form of synchronizer is not guaranteed to work correctly in combination with any arbitrary synchronous protocol [15]. However, we showed that the level of synchronization provided by this synchronizer is sufficient for the problem at hand. The synchronous protocol given in [13] for finding centers and medians in a network is another one that can take advantage of this form of synchronizer. Also, observe that for any graph problem on a network, it is always possible to construct an algorithm which routes all topological information to a single site and then uses sequential techniques for solving the problem. Such a "degenerate" algorithm will also have a worst case message complexity of \( O(mn) \), but will not be considered "truly" distributed [2].

Appendix

FORMAL PRESENTATION OF THE ALGORITHM

Messages of the Algorithm

| LENGTH(i) | Sent to inform a successor node of a path of total weight \( t \). |
| ACK | Sent to inform a predecessor that the LENGTH message sent by that node has been processed. |
| WAKEUP | Sent to all neighbors at the beginning of the computation. |
| GO | Sent to all neighbors, permitting them to execute the actions corresponding to the next pulse. |
| TERMINATE(b) | Sent to successor nodes to terminate distance updating activity. The Boolean variable \( b \) is true if the sender node is reachable from a negative cycle; it is false otherwise. |

Variables Kept at Node \( i \)

| \( d(i) \) | Current estimate of the distance, i.e., weight of the shortest path from the source node. Initially, \( d(i) \) is set to \( \infty \) for all nodes. |
| \( \text{pred}(i) \) | Predecessor node in the current shortest path. Initially, \( \text{pred}(i) = i \) for all nodes. Finally, \( \text{pred}(i) = i \) only for the source node \( s \). |
| \( \text{num}(i) \) | Number of unacknowledged messages at node \( i \). Initially, \( \text{num}(i) = 0 \) for all nodes. |
| \( \text{change}(i) \) | A Boolean variable set to \( \text{true} \) if \( d(i) \) changed since the last sending of LENGTH messages; it is \( \text{false} \) otherwise. Initially, \( \text{change}(i) = \text{false} \) for all nodes. |
| \( \text{negcycle}(i) \) | A Boolean variable set to \( \text{true} \) if node \( i \) is reachable from a negative cycle; it is \( \text{false} \) otherwise. Initially, \( \text{negcycle}(i) = \text{false} \) for all nodes. |
| \( \text{awake}(i) \) | A Boolean variable set to \( \text{true} \) if a WAKEUP message has been received and a TERMINATE message has not been sent or received; it is \( \text{false} \) otherwise. Initially, \( \text{awake}(i) = \text{false} \) for all nodes. |
| \( \text{visited}(i) \) | A Boolean variable set to \( \text{true} \) if a TERMINATE message has been received; it is \( \text{false} \) otherwise. Initially, \( \text{visited}(i) = \text{false} \) for all nodes. |
| \( \text{neighbors}(i) \) | Set of neighbors of node \( i \) (input). |
| \( \text{successors}(i) \) | Set of successors of node \( i \) (input). |
| \( s(i) \) | Number of successors of node \( i \) (input). |
| \( \text{ackset}(i) \) | Subset of predecessor nodes to whom ACK messages have to be sent. Initially, \( \text{ackset}(i) \) is set to \( \text{empty} \) for all nodes. It should be implemented as a multiset. |
| \( \text{goreceived}(i) \) | Subset of neighbors from whom GO messages have been received. Initially, \( \text{goreceived}(i) \) is set to \( \text{empty} \) for all nodes. It should be implemented as a multiset. |

To trigger the algorithm, the source node delivers a WAKEUP message, followed by a LENGTH(0) message to itself.

Algorithms at Node \( i \)

for LENGTH(i) message from \( j \) do

begin

if \( t < d(i) \) then [if it is source node, detect negative cycle]

if \( \text{pred}(i) = i \) and \( d(i) = 0 \)

then begin negcycle(i) := true;

execute procedure phase2

end

else begin [generate ACK for current predecessor, if not done so already]

if \( \text{num}(i) > 0 \) or \( \text{change}(i) = \text{true} \)

then include \( \text{pred}(i) \) in \( \text{ackset}(i) \);

[update shortest path]

\( d(i) := t; \text{pred}(i) := j \);

[if there are successors, changed distance should be propagated]

if \( s(i) = 0 \)

then include \( \text{pred}(i) \) in \( \text{ackset}(i) \)
else change(i): = true
end
else [new length does not denote a shorter path]
include j in ackset(i)
end
end

for ACK message from j do
begin
[decrement the count of unacknowledged LENGTH messages]
num(i): = num(i) - 1;

if all LENGTH messages sent have been acknowledged and
no new distance update took place, generate ACK to pred-
ecessor, provided one exists]
if num(i) = 0 and change(i) = false
then [if it is source node, start the second phase]
if pred(i) = i and d(i) = 0
then begin negcycle(i): = false;
execute procedure phase2
end
else include pred(i) in ackset(i)
end

procedure newpulse;
begin
[send ACK messages]
for all k ∈ ackset(i)
send ACK to k;
ackset(i) : = φ;

[send LENGTH messages to all successors, if necessary]
if change(i) = true
then begin for all k ∈ successors(i)
send LENGTH(d(i) + w(i, k)) to k;
num(i) : = num(i) + 1;
change(i): = false
end
end

for WAKEUP message from j do
begin
if awake(i) = true
then ignore the message
else begin awake(i): = true;
[propagate WAKEUP]
for all k ∈ neighbors(i)
send WAKEUP to k;
[propagate GO to start first pulse]
for all k ∈ neighbors(i)
send GO to k
end
end

for GO message from j do
begin
include j in goreceived(i);

[if goreceived(i) ⊇ neighbors(i) and awake(i) = true]
then begin goreceived(i): = goreceived(i) - neighbors(i);
execute procedure newpulse;
[propagate GO to start next pulse]
for all k ∈ neighbors(i)
send GO to k
end
end

procedure phase2;
begin
awake(i): = false;
if negcycle(i) = true
then for all k ∈ successors(i)
send TERMINATE(true) to k
else for all k ∈ successors(i)
send TERMINATE(false) to k
end

for TERMINATE(b) message from j do
begin
awake(i): = false;
if b = true
then [reachable from a negative cycle]
if negcycle(i) = true
then ignore the message
else begin negcycle(i): = true;
for all k ∈ successors(i)
send TERMINATE(true) to k
end
else [check for negative cycle, if not done so already]
if visited(i) = true
then ignore the message
else begin if num(i) > 0 or change(i) = true
then begin negcycle(i) : = true;
for all k ∈ successors(i)
send TERMINATE(true) to k
end
else for all k ∈ successors(i)
send TERMINATE(false) to k
end;
[record that at least one TERMINATE message has been processed]
visited(i): = true
end

Acknowledgment
The authors thank the anonymous referees for suggesting several improvements to the presentation.

References
[15] K. B. Lakhman and K. Thulasiraman, "On the use of synchroniz-
An Efficient Distributed Knot Detection Algorithm

ISRAEL CIDON

Abstract—A distributed knot detection algorithm for general graphs is presented. The knot detection algorithm uses at most $O(n \log n + m)$ messages and $O(m + n \log n)$ bits of memory to detect all knots' nodes in the network (where $n$ is the number of nodes and $m$ is the number of links). This is compared to $O(n^2)$ messages needed in the previous published best algorithm. The knot detection algorithm makes use of efficient cycle detection and clustering techniques.

Various applications for the knot detection algorithms are presented. In particular, we demonstrate its importance to deadlock detection in store and forward communication networks and in transaction systems.

Index Terms—Clustering, cycle detection, deadlock detection, distributed algorithms, knot detection.

I. INTRODUCTION

A knot in a directed graph is a strongly connected subgraph with no edge directed away from the subgraph. A knot is a useful concept for describing deadlocks in computer systems. In [1], [2], deadlocks in store and forward networks are described as knots in the buffer graph of the network. The concept of a knot in the buffer graph is also used for deadlock resolution techniques when the deadlock is resolved by discarding packets at nodes. The minimum number of packets that should be discarded in order to resolve the deadlock is exactly one packet in each knot [3], [4]. Knots were also found to be useful for representing deadlocks in transaction systems [5], [6].

A distributed knot detection algorithm is the basis for developing a distributed deadlock detection algorithm. In [3], a deadlock detection algorithm for buffer deadlock in store and forward networks is described which is based on a knot detection for a general graph. The algorithm developed here can replace the knot detection of [3], resulting in a more efficient deadlock detection algorithm.

Various distributed knot detection algorithms have been suggested in the literature. Some of them are imbedded in more general deadlock detection algorithms. Three basic classes of algorithms have been suggested.

1) Collecting the complete graph topology at each node and detecting the knot by each individual node [2].
2) Testing individually at each node whether it is a member of a knot using a search algorithm. A distinct search is used for each node [4]-[6].
3) Using cycle detection and clustering technique in which cycles of clusters are detected and merged into bigger clusters [3].

II. THE MODEL

A network consists of a set of communication nodes $N$ and a set of bidirectional communication links $L$ that interconnect the nodes of $N$.

Regarding links, the following properties are assumed. They are FIFO (do not lose, reorder, or duplicate messages); there is no bound on the amount of time that it takes a message to traverse a link; any message placed on the link arrives at the other side of the link in finite time; links never fail.

We assume that at each node $i$, each attached link $l$ may be designated as an outgoing link. (In deadlock detection, this implies that there is a request pending for this specific link.)

Let $(V, E)$ be a directed graph where $V = N$ is the set of vertices in the graph and $E$ a set of directed edges where a directed edge $(i, j)$ indicates that in node $i$ the link $(i, j)$ is designated as an outgoing link. A tie in $(V, E)$ is a set of nodes with no links directed from $T$ to $N - T$.

A knot $K$ is a tie of which any subset is not a tie. This implies that $K$ is a set of strongly connected nodes. Alternatively, node $i$ is a member of a knot if $i$ is reachable from all nodes which are reachable from $i$. In that case, the knot is the set of nodes which are reachable from $i$ (including $i$ itself). Obviously, any tie contains at least one knot. In Fig. 1, an example for a knot and ties is depicted.