January 1982 LIDS-P-1175

DISTRIBUTED MINIMUM HOP ALGORITHMS*

by

Robert G. Gallager**

* This research was conducted at the M.I.T. Laboratory for Information and Decision Systems with partial support provided by Defense Advanced Research Projects Agency under Contract No. ONR/N00014-75-C-1183 and National Science Foundation under Contract No. NSF/ECS 79-20834.

**Room No. 35-206, Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, Cambridge, Massachusetts 02139.

Distributed Minimum Hop Algorithm

I. Introduction

The control of data communication networks (and any other large distributed systems) must be at least partly distributed because of the need to make observations and exert control at the various nodes of the network. When one also considers the desireability of letting networks grow (or shrink due to failures), it is reasonable to consider control algorithms with minimal or no centralized operations.

In developing distributed algorithms for such control functions as routing and flow control, for example, it becomes evident that there are a number of simple network problems which arise repeatedly; distributed algorithms for solving these simple problems are then useful as building blocks in more complex algorithms. Some of these frequently occuring simple problems are as follows: a) the shortest path problem--given a length for each edge in the network, find the shortest path between each pair of nodes (or the shortest path between one given node and each other node); b) the minimum hop problem, which is a special case of the shortest path problem in which each edge has unit length; c) the minimum spanning tree problem--given a length for each edge (undirected), find the spanning tree with the smallest sum of edge lengths; alternatively, for directed edges, find the routed minimum length directed spanning tree for each root; d) the leader problem--find the node in the network with the smallest ID number; e) the max flow problem--given a capacity for each edge, find the maximum traffic flow from a given source node to a given destination.

We now must be more precise about our assumptions concerning distributed algorithms. Mathematically we model the network as a connected undirected graph with, say, n nodes and e edges. Each node contains the facilities for doing computations, storing data, and sending and receiving messages over the adjoining edges. Messages are assumed to be transmitted without error but with an unknown variable finite delay. Successive messages in a given direction on a given edge are queued for transmission at the sending node, are transmitted, arrive in the order transmitted and are queued waiting for processing by the receiving node.

Initially each node stores only the unique identity of the node itself and the relevant parameters such as length or capacity of its adjacent edges. The computational facility at each node executes a local algorithm that specifies initial operations to start the algorithm and the response to each received message. These initial operations and responses include both computation and sending messages over adjacent edges. A distributed algorithm is the collection of these local algorithms used for solving some global problem such as the building block problems mentioned above.

For readers familiar with layered network architectures, the assumptions above effectively assume the existence of lower level line protocols. Error detection and retransmission provides the error free but variable delay transmission, and message formatting provides the ability to receive and process messages as entities. Our algorithms will contain no interrupts, no time outs, and will be independent of particular hardware or software constructs. Our assumptions also effectively preclude the possibility of node and edge failures. This is not because the problem of failures is unimportant, but rather because we feel tht a more thorough understanding of failure free distributed algorithms is necessary before further progress can be made on the problem of failures.

Since communication is often more costly than computation in a network, reasonable measures of complexity for distributed algorithms are the amount of communication required and the amount of time required. We measure communication complexity C in terms of the sum, over the network edges, of the number of elementary quantities passed in messages over those links. The elementary quantities are node identities, edge lengths, capacities, number of nodes or edges in certain subsets, and so forth. These quantities could be easily translated into binary digits, but this is usually unnecessary. The time complexity, T of an algorithm is the number of units of time required if each communication of an elementary quantity over an edge requires one time unit and computation requires negligible time. This is under the proviso that the algorithm must continue to work correctly when communication requires uncertain time.

There is one significant difficulty with using communication and time as the measures of goodness of distributed algorithms. It is very easy simply to send all the topological information about the network to each of the nodes and then solve the problem at each node in a centralized fashion. We shall see shortly that the communication complexity of this approach is O(ne) where n is the number of nodes in the network and e is the number of edges. By O(ne), we mean there is a constant c such that for all networks, the required number of communications is at most cnE.

Similarly the time required in this approach is $O(e)$. One could argue that in some sense this approach is not really distributed, but it turns out to be non trivial to find "really distributed" algorithms that do any better than $C = O(ne)$ and $T = O(e)$.

One known example [1] of a distributed algorithm better in communication and time than the above centralized approach is a shortest spanning tree algorithm with $O(e+n \log n)$ communication and $O(n \log n)$ time. In this paper, our major concern is algorithms for finding the minimum hop paths from all nodes of a network to a given destination. First we describe four rather trivial algorithms. The first three find the minimum hop paths between all pairs of nodes. The best of these, in terms of worst case communication and time, uses $O(en)$ communication and $O(n)$ time; the fourth algorithm finds minimum hop paths to a single destination with $O(n^2)$ communication and $O(n^2)$ time. Both algorithms have a product of time and communication, in terms of n alone, of $O(n^4)$. We then develop a class of less trivial algorithms providing an intermediate tradeoff between time and communication; one version of this yields $O(n^{1.5})$ time and $O(n^{2.25})$ communication, for a time communication product of $O(n^{3.75})$.

It is surprising at first that the minimum hop problem appears to be so much more difficult (in terms of time and communication) than the minimum spanning tree problem. The reason for this appears to be that the minimum spanning tree has local properties not possessed by the shortest hop problem; for example the minimum weight edge emanating from any given node is always in a minimum spanning tree.

All of our subsequent results will be in terms of the orders of worst case communication complexity and time complexity of various algorithms. One should be somewhat cautious about the practical interpretation of these results. In the first place, the results are worst case and typical performance is often much better. In the second place, for most present day packet networks there is a large overhead in sending very short control messages, so that algorithms that can package large amounts of control information in a single packet have a practical advantage over those that cannot accomplish this packaging.

II. Some Simple Minimum Hop Algorithms

In this section we shall develop three simple global minimum hop algorithms for finding a minimum hop path between each pair of nodes in the network and then a single destination algorithm for finding minimum hop paths from each node to a given destination node. It should be clear that a single destination algorithm could be repeated n times, once for each destination, to solve the global problem, and that the global problem automatically solves the single destination problem. We could also look at the problem of finding just the minimum hop path between a given pair of nodes, but we conjecture that this problem, in the worst case, is no easier than the single destination problem.

At the initiation of any of these algorithms, each node knows its own identity and its number of adjacent edges. At the completion of the algorithm, for the single destination case, each node other than the destination knows its hop length to the destination and has identified a single adjacent edge, called its inedge, as being on a minimum hop path to the destination. It is not necessary for a given node to know the entire minimum hop path to the destination, since a node can send a message to the destination over a minimum hop path by transmitting it on its inedge. The neighboring node can then forward the message over its inedge and so forth to the destination. For the global minimum hop problem, each node, at the completion of the algorithm contains a table with one entry for each node in the network. Such an entry contains the node identity, the hop level, which is the number of hops on a minimum hop path to that node, and the inedge, which is the adjacent edge on such a minimum hop path. Initially a node's table contains only a single entry containing the node itself, at a hop level of 0 and a null inedge.

Algorithm G1

To begin, we give an informal description of a global minimum hop algorithm* that, for each node, first finds all nodes at hop level 1, then hop level 2, and so forth. Initially each node is in a "sleeping" state. To start the algorithm, one or more nodes "wakeup" and send a message containing their own identity over each of their adjacent edges. When a sleeping node receives such a message, it "wakes up" and sends its own identity over each adjacent edge. Each node, sleeping or not, which receives such a message, places the received node identity in a table, identifying the inedge to that destination as the edge on which the message was received, and identifying the hop length as one.

When a node receives the above identity message over each of its adjacent edges (which must happen eventually), it then knows the identity of each of its neighboring nodes. It then sends a message called a "level 1" message over each adjacent edge, listing the identities of each of these neighboring nodes. When a node receives a level one message over an adjacent edge, each of the received node identities that is not already in its table at hop level 2 or less is added to the table, identifying the inedge as the edge on which the message was

*This algorithm was developed by the author six years ago as part of a failure recovery algorithm. It has undoubtedly been developed independently by others.

received and identifying the hop level as two. When a node has received "level 1" messages over each adjacent edge, its table contains all nodes up to two hops away, and it sends a "level 2" message over each adjacent edge containing a list of all node identities two hops away. In general, for $\ell \geq 1$, when a node receives a "level ℓ" message over an adjacent edge, each received node identity not already in its table at hop level $\ell + 1$ or less is added to the table, identifying the inedge for that destination as the edge on which the message was received and identifying the hop level as $\ell + 1$. After receiving "level ℓ" messages over each adjacent edge, the node transmits a "level $\ell + 1$" message containing a list of all nodes $\ell + 1$ hops away. Finally, if this list of nodes $\ell + 1$ hops away is empty, the node knows that its table is complete and its part in the algorithm is finished. Neighboring nodes need take no special account of such an empty list since they must also finish before looking for level $\ell + 2$ messages.

The communication complexity of this algorithm is easily evaluated by observing that each node identity is sent once over each edge in each direction, leading to the communication of 2ne identities in all, or $C = O(ne)$. Including the level numbers of the messages does not change this order of communication. The time complexity can be upper bounded by observing that messages must be sent at a maximum of n levels and at most n time units are required for each level; also at most n time units are required for the initial transmission of identities. Thus we have $T = O(n^2)$. By considering the dumb-bell network of Figure 1, it can be seen that $O(n^2)$ time units are actually required in the worst case.

Algorithm G2

Strangely enough, a slight modification of algorithm G1 reduces T to $O(n)$ while not changing the order of C. In algorithm G1, for each level ℓ, each node sends a single level ℓ message over each adjacent edge containing the list of all nodes at hop level ℓ. In the modification, algorithm G2, this single level ℓ message is broken into a set of shorter messages, one for each node at hop level ℓ and a message to indicate the end of the list. After the node receives the end of list $\ell-2$ message on each edge, it can send its own end of list $\ell-1$ message, and immediately after transmitting that message, it can start to transmit any level ℓ messages already in its table. One can qualitatively see the time saving in Figure 1, where the cliques of nodes at either end of the dumb-bell structure will have their identities transmitted in a pipelined fashion over the area in the center of the dumb-bell, rather than being transmitted in totality on a single edge at a time. A proof that $T = O(n)$ is given in the Appendix.

Algorithm G3

Another simple modification of this algorithm corresponds to the distributed form of Bellman's algorithm; it is essentially the shortest path routing algorithm used in the original version of the Arpanet algorithm [2], specialized to minimum hops. The table of node identities, inedges, and hop numbers operates as before, but whenever a node is added to the table or the hop number is reduced, that information is immediately queued for transmission on each adjacent edge. Because of this overeagerness to transmit, the same node identity can be transmitted up to n-2 times over the same edge, each time at a smaller hop number. This increases the worst case communication complexity to $C = 0(n^2e)$. If the transmission queues at the nodes are prioritized to send smaller hop numbers before larger hop numbers, the time complexity can be shown to be $T = 0(n)$. Although this distributed form of Bellman's algorithm is quite inferior in terms of worst case communication complexity, its typical behavior is quite good and its lack of waiting for slow transmissions is attractive from a system viewpoint. This type of algorithm can also be applied to a much more general class of problems than minimum hop, and general results concerning the convergence of such algorithms have recently been established [3].

Another advantage of the distributed Bellman algorithm is that it can easily be modified for the single destination minimum hop problem. In this case, only the destination node need be communicated through the network and it is easy to see that the communication complexity is now reduced to $C = 0(ne)$, while T remains at $0(n)$. This is of course a worst case result corresponding to a highly pathological pattern of communication delays. In typical cases, this algorithm would require little more than one communication over each edge in each direction, but it is still of theoretical interest to see whether algorithms exist with worst case communication complexity less than $0(ne)$. What is required is somewhat more coordination in the algorithm to prevent a node from broadcasting a large hop length to the destination when that node would shortly find out that it is much closer to the destination. The next algorithm, called the coordinated shortest hop algorithm is designed to provide this coordination through the destination node.

Algorithm D1

The coordinated shortest hop algorithm works in successive iterations, with the beginning of each iteration synchronized by the destination node. On the ith iteration, the nodes at hop level i from the destination discover they are at level i, choose an inedge, and collectively pass a message to the destination that the iteration is complete. To be more precise, at the end of the i-1 iteration, there is a tree from the destination node to all nodes at level i-1. At the beginning of iteration i, the destination node broadcasts a message on this tree to find the level i nodes. Each node at level $j < i-1$ receives this message on its inedge and sends it out on each of its adjacent edges that are in the tree (other than the inedge). Each node at level i-1 receives the message on its inedge and sends out a test message on each edge not already known to go to a lower level node. When a node not at level $j < i$ first receives the test message, it designates itself as level i, designates the edge on which the test message was received as the inedge, and acknowledges receipt to the sender with an indication that the link is its inedge. On subsequent receptions of a test message, the node acknowledges receipt with an indication that the edge is not its inedge.

When a node at level i-1 receives acknowledgements over each edge on which it sent a test message, the node sends an acknowledgement over its own inedge, indicating also how many level i nodes can be reached through the given node. When a node at level j, $0 < j < i-1$, receives acknowledgements over all outgoing edges in the tree, it sends an acknowledgement over its inedge, also indicating how many level i nodes can be reached through itself (i.e. adding up the numbers from each of its outgoing edges). Finally, when node d receives an acknowledgement over each edge, the level i iteration is complete. If any level i nodes exist (which is now known from the numbers on the acknowledgement), node d starts iteration i+1, and otherwise the algorithm terminates.

A detailed description of the algorithm is given in pidgin algol in the appendix. It would be helpful to understand this algorithm as a prelude to the more refined algorithm of the next section. The communication complexity of the algorithm can be determined by first observing that at most one test message is sent over each edge in each direction. Also at most n coordinating messages are sent over each edge in the minimum hop tree. Since there is one acknowledgement for each coordinating or test message, the total number of messages is bounded by $2(n^2 + e)$, or $C = 0(n^2)$. Since the coordinating messages travel in and out over the tree in serial fashion, we also have $T = 0(n^2)$.

III. A Modified Coordinated Minimum Hop Algorithm-Algorithm D2

We saw in the last section that the coordinated shortest hop algorithm was quite efficient in communication at the expense of a large amount of required time. Alternatively, the distributed Bellman algorithm is efficient in time but poor in communication. Our objective in this section is to develop an algorithm providing an intermediate type of tradeoff between time and communication.

One would imagine at first that the coordinated algorithm would work very efficiently for dense networks in which the longest of the shortest hop paths are typically small, thus requiring few iterations, whereas the Bellman algorithm would be efficient for very sparse networks. Figure 1, however, shows an example where both algorithms work poorly for the worst case of communication delays; in this example the coordinated algorithm requires $0(n^2)$ times and $0(n^2)$ communication and the Bellman requires $0(n)$ time and $0(n^3)$ communication.

In the approach to be taken here, we effect a trade-off between time and communication by coordinating the algorithm in groups. In the first group, we find all nodes that are between one hop and k_1 hops from the destination, where k_1 is an integer to be given later. In the second group, all nodes between k_1+1 and k_2 hops away are found, and in general, in the gth group, all nodes between $k_{g-1}+1$ and k_g hops from the destination are found.

At the beginning of the calculation of the gth group, a shortest hop tree exists from the destination to each node at level k_{g-1}. A global synchronizing message is broadcast from the destination node out on this tree to coordinate the start of the calculation for this group, and at the end of the group calculation, an acknowledgement message is collected back in this tree to the destination. This broadcast and collection is essentially the same as that done for each level in the coordinated algorithm of the last section. The saving of time in this modified algorithm over the coordinated algorithm is essentially due to the fact that these global coordinating messages are required only once per group rather than once per level.

Within the gth group of the algorithm, the search for nodes at levels $k_{g-1}+1$ to k_g is coordinated by a set of nodes called synch nodes; these nodes are at level k_{g-2}. Each synch node for group g is responsible for coordinating the search for the nodes at levels k_{g-1} to k_g whose shortest hop path to the destination passes thru that synch node. Each synch node essentially uses the coordinated shortest hop algorithm of the last section (regarding itself as the destination) to find successively nodes at level k_{g-1}, then $k_{g-1}+2$, and up to k_g. After completing the process out to level k_g, the acknowledgements are then collected all the way back to the destination node, which then initiates the next group.

There is a complication to the above rather simple structure due to the fact that there is no coordination between the different synch nodes for a given group. Thus one synch node might find a node that appears to be at level ℓ according to the tree being generated from that synch node, and later that same node might be found at some level $\ell' < \ell$ in the tree generated more slowly by another synch node. What happens in this case is that the node so effected changes its level from ℓ to ℓ' and changes its ingoing edge from the first tree to the second tree. The situation is further complicated by the fact that the given node might be helping in the first synch node's search for nodes at yet higher levels, and the change in the node's inedge cuts off an entire portion of the tree generated by the first synch node. The precise behavior of the algorithm under these circumstances is described by the pidgin algol program in the appendix which is executed by each node. The following more global description, however, will help in understanding the precise operation.

Each node has a local variable called _level_ giving its current estimate of the number of hops to the destination. A node's level is set by a test message coming via synch messages from some synch node. If part of a tree is cut off, as in the example above, the node whose inedge is changed immediately changes its level, but the more distant nodes remain in an inconsistent state for a while, changing their levels later in response to new test messages.

As in the coordinated algorithm, each node maintains a state for each adjacent edge, the possible states being _unused_, _in_, _active_, and _inactive_. Active and inactive are outgoing edges in the current estimate of the evolving minimum hop tree, inactive indicating that no new nodes were found using that edge in the last iteration of the algorithm.

An important property of the algorithm is that each test or synch message sent over an edge is later acknowledged by precisely one ack message bearing the same level as the message being acknowledged. Normally the acknowledgements are sent in the same way as in the coordinated algorithm. When a node changes level, however, it immediately sends the acknowledgement to any yet unacknowledged synch message. Similarly if a synch message arrives on the old in edge after a level change, that is acknowledged immediately. In both cases, the acknowledgement carries the level of the corresponding synch and indicates that no new nodes have been found at that level, thus changing the opposite node's edge state to inactive (assuming the opposite node has not also changed levels in the meantime). The inactive edge state at the opposite node is inappropriate, but the given node must later send a test message over that edge, removing the temporary inconsistency.

The next important property of the algorithm has to do with the ability of a node to determine which test or synch message a given acknowledgement corresponds to. The problem is that a node might change levels and subsequently send out test messages corresponding to its new tree before receiving acknowledgements from messages corresponding to its old tree. A node may, in fact, change levels several times before receiving these old acknowledgements. The property is that each acknowledgement to an old message on a given edge must be received before receiving the acknowledgement to the test message corresponding to the node's new level. Suppose now that ℓ was an old level of the node when some test or local synch message was transmitted and ℓ' is the new node level. We must have $\ell' < \ell$, since a node only changes level when it finds a shorter hop path through another synch node. The new test message that the node subsequently transmits after changing to level ℓ' is at level $\ell'+1$, so we have $\ell'+1 \leq \ell$. Since any old test message or synch message is at a level greater than ℓ, and these acknowledgements are received before the acknowledgement of the new test message, the node can distinguish the old ack messages from the new by the level numbers.

The final property of the algorithm, which follows from the previous description, is that the time required for a test or synch message to be acknowledged is at most the time required for the message to be broadcast out to the appropriate level and be collected back again. In other words, the fact that some of the nodes further out in the tree may have changed levels and joined another tree does not increase the acknowledgement time. With these properties, the reader should be able to convince himself of the correct operation of the algorithm.

We now turn our attention to the destination node and the calculation of the number of levels in each group. At the completion of the $(g-1)^{th}$ group, the shortest hop tree has been formed out to level k_{g-1} and the acknowledgement of this fact has arrived back at the destination. For reasons that will be apparent later, the choice of level k_g depends on the number, m_{g-2}, of nodes at level k_{g-2} that will synchronize the next group. To find this number, the destination synchronizes a single global iteration of the algorithm at level $k_{g-1}+1$ (i.e., the first level of the new group). The acknowledgements to this special iteration are arranged to count the number of nodes at level k_{g-2} that still have active edges at this iteration. More specifically, the destination broadcasts the message (synch k_{g-2}). This is broadcast on the active edges of the current minimum hop tree out to level k_{g-2}. The nodes at this level go into a state called "Presynch" and broadcast the message (synch k_{g-1}+1) on their active edges. This message in turn propagates outward, extends the tree to level k_{g-1}+1, and the acknowledgements return to the presynch nodes. Any presynch node with active edges at this point goes into a state called "Synch" and acts as a synch node for the next group. The number of acknowledgements from these newly formed Synch nodes are then collected back to the destination, yielding the number m_{g-2} of synch nodes.

The rule for calculating k_g is now as follows:

$$k_g = \begin{cases} k_{g-1} + \lceil n^{2x} \rceil & \text{if } m_{g-2} < n^x \\ k_{g-1} + \lceil n^x \rceil & \text{if } m_{g-2} \geq n^x \end{cases} \tag{1}$$

where n is the number of nodes in the network, x is a parameter, and $\lceil y \rceil$ is the smallest integer greater than or equal to y. The reason for this rather peculiar rule will be evident when we calculate the communication and time complexity of the algorithm. We observe however, that the root must know the number of nodes in the network in order to use this rule. A simple distributed algorithm for the root to calculate n is given in the Appendix. In this algorithm, the destination node broadcasts a test message throughout the network, and a directed spanning tree, directed toward the destination, if formed by each node choosing its in edge to be the edge on which the test message is first heard. The number of nodes is then accumulated through this spanning tree. The algorithm sends two messages over each edge and requires $0(n)$ time, so it does not effect the order of communication or time required by our overall algorithm.

Before finding the number of messages and time required by the modified coordinated shortest hop algorithm, we first find an upper bound on the number of groups. From (1), the groups are divided into big groups, with $\lceil n^{2x} \rceil$ levels and little groups with $\lceil n^{x} \rceil$ levels. Each big group (except perhaps the final one) contains at least one node at each level, or at least n^{2x} nodes. Thus there are at most $n^{1-2x}+1$ big groups. For each little group, say group g, the preceding group contains at least n^{2x} nodes. To see this, we observe from (1) that $m_{g-2} \geq n^{x}$, which means that at the conclusion of generating the minimum hop tree out to level $k_{g-1}+1$, there are at least n^{x} nodes at level k_{g-2} that have paths out to k_{g-1} in that minimum hop tree. Each node path includes at least n^{x} nodes, yielding the result. Thus there are at most n^{1-2x} little groups, so the total number of groups G satisifes

$$G \leq 2n^{1-2x} + 1 \qquad (2)$$

A more refined analysis, which is unnecessary for our purposes, reduces this bound to n^{1-2x}.

Observe now that a node can receive at most two globally coordinated synch message for each group, so the total number of these messages is at most 2nG. Next note that a node in the gth level can only change levels during the computation of the gth group and, that the levels can only decrease, so that $k_g - k_{g-1} -1$ upper bounds the number of level changes. Note also that each level adopted by a node corresponds to the shortest hop path through a given synch node, so that m_{g-2} also upper bounds the number of level changes. From (1), either $m_{g-2} \leq n^x$ or $k_g - k_{g-1} - 1 \leq n^x$. In summary, a node can send out test messages at most n^x times, so that each edge can carry at most n^x test messages in each direction, and $2e\ n^x \leq n^{2+x}$ is an upper bound on the total number of test messages.

Finally we must consider the number of locally coordinated synch messages. If a node in group g is initially found at level ℓ_1, then that node can receive at most $k_g - \ell_1 \leq n^{2x}$ local synch messages before either the node changes level again or the group computation is completed. Similarly after ith level change at that node to level ℓ_i, say, at most $k_g - \ell_i$ local synch messages arrive before the next level change or group completion. Finally after the group completion, the node receives at most $k_{g+1} - k_g -1 \leq n^{2x}$ local synch messages for the next group. Adding up these terms, we find that at most $n^{3x} + n^{2x}$ local synch messages arrive at each node, and $n^{1+3x} + n^{1+2x}$ is an upper bound to the total number of locally coordinated synch messages.

Adding all of these types of messages together, and recalling that there is one acknowledgement message for each other message, the total number of messages, required on the algorithm is upper bounded by

$$C \leq 2[(2n^{2-2x} + n) + 4en^{x} + (n^{1+3x} + n^{1+2x})] \quad (3)$$

$$\leq 4n^{2-2x} + 2n + 4n^{2+x} + 2n^{1+3x} + 2n^{1+2x} \quad (4)$$

where (3) provides a bound in terms of e and n and (4) provides a bound in terms of n alone. We can express (4) as

$$C = O(n^{c}) \; ; \quad c = \max(2 + x, 1 + 3x) \quad (5)$$

A similar analysis of required time can now be carried out; we first calculate an upper bound on the time for a single group. The globally coordinated synchs and their acknowledgements clearly take at most time 4n. Each synch node must then send out at most n^{2x} local synch messages. The time required for one of these local synch messages to propagate out to the intended level (with a final test message at the outermost level), and then be acknowledged back to the synch node n, is at most $4 \lceil n^{2x} \rceil$ (since each group contains at most $\lceil n^{2x} \rceil$ levels and the local synch messages for group g also traverse the nodes in group g-1). Since the different synch nodes within a group operate in parallel and the communications initiated by one never have to wait for those of another, the total time required by a group is at most $4n + 4 \lceil n^{2x} \rceil \lceil n^{2x} \rceil$. Thus the overall required time is this quantity times G, or

$$T \leq [2n^{1-2x} + 1][4n + 4 \lceil n^{2x} \rceil^{2}] \quad (6)$$

$$T = O(n^{t}) \; ; \; t = \max(2 - 2x, 1 + 2x) \quad (7)$$

We see that t in (7) is minimized at $t = 3/2$ by $x = 1/4$, which leads (from (5)) to $c = 9/4$. We also see that with this algorithm, there is a tradeoff between communication and time. By reducing x from 1/4 toward 0, t increases linearly from 3/2 to 2, whereas c decreases linearly from 9/4 to 2.

The above results are in terms of only the number of nodes, using the bound $e \leq \frac{1}{2} n^2$ on edges. For relatively sparse networks, the required amount of communication is considerably reduced. Let us define α by $e = n^{1+\alpha}$, so that α close to zero corresponds to sparse graphs and α close to 1 corresponds to dense graphs. With this modification, the required time is still given by (7) and the number of messages is now given by

$$C = 0(n^c) \quad ; \quad c = \max(2 - 2x,\ 1 + \alpha + x,\ 1 + 3x) \qquad (8)$$

It is not difficult to see that both t and c increase with x for $x \geq 1/4$, so the tradeoff region of interest is $0 < x \leq 1/4$. Figure 2 shows the resulting tradeoff between t and c for different values of the sparseness parameter α. For each α, as t is increased from 3/2, c decreases, but cannot go below t. It should be noted that all values of α less than 0.4 have the same tradeoff between t and c, going from $t = 1.5$, $c = 1.75$ to $t = 1.6$, $c = 1.6$.

The careful reader will note that not much use can be made of the above tradeoffs unless the sparseness parameter α is known. It is an easy matter, though, to evaluate e as part of the algorithm for evaluating n given in the Appendix.

Appendix

Proof that algorithm G2 has T = 0(n)

Since each message contains only a single node identity or level number, we assume for the purposes of the proof that a message whose actual transmission starts at time t is completely received by time t+1. Thus, if the first node to wake up starts to transmit its identity by time 0, then its neighbors will wake up and start to transmit their identities by time 1, and by time n each node will have heard the identity of each of its neighbors, and will have transmitted the message indicating the end of its "list of nodes zero hops away". These two facts form the basis for the inductive argument to follow.

Let $N_i(x)$ be the order in which node i transmits the identity of node x in the algorithm. Thus $N_i(i) = 1$ since i transmits its own identity first; for the first neighbor, x, in i's list of neighbors, $N_i(x) = 2$, and so forth. We assume without loss of generality that each node i transmits the identities of nodes ℓ hops away in the order in which it heard about them.

Lemma 1: Assume that node i's table entry for node x has an in edge going to node j. Then $N_i(x) \geq N_j(x)$.

Proof: Let y be any node for which $N_j(y) < N_j(x)$. Then node i received node j's hop length to y before receiving node j's hop length to x. Thus, whether or not node i's minimum hop path to x goes through j, node i will transmit the identity of y before that of x. Now y in the above argument can be chosen as any of the $N_j(x)-1$ nodes for which $N_j(y) < N_j(x)$. Since i transmits each of these before x, $N_i(x) \geq N_j(x)$.

Let $N_i'(\ell)$ be the number of nodes ℓ or fewer hops away from i, with $N_i'(0) = 1$. Note that if a node x is ℓ-1 hops away from some node j, it can be at most ℓ hops away from a neighbor i of node j. Thus we have the following lemma.

Lemma 2: $N_j'(\ell-1) \leq N_i'(\ell)$ for $\ell \geq 1$.

Let $\tau_i(x)$ be the time at which node x is transmitted by node i (assuming the algorithm starts at time 0 and that each transmission takes at most unit time) and let $h_i(x)$ be the hop level of x from i. Let $\tau_i'(\ell)$ be the transmission time of the message indicating the end of transmissions at hop level ℓ. The following lemma now shows that $T = 0(n)$ (and in fact that $T < 3n$).

Lemma 3: For each pair of nodes i and x,

$$\tau_i(x) \leq n-2+h_i(x) + N_i(x) \tag{A1}$$

$$\tau_i'(\ell) \leq n-1+\ell+N_i'(\ell) \tag{A2}$$

Proof: We use induction on the right hand side of (A1) and (A2), using n-1 as the basis for (A1) and n as the basis for (A2). The right hand side of (A1) is n-1 only for x = i, and (A1) has been established for this case. The right hand side of (A2) is n only for $\ell = 0$, which is established, and the right hand side of (A1) is never n. For the induction step, first consider (A1) for an arbitrary i,x. In order for node i to transmit x by the given time, three conditions are necessary. First, any y for which $N_i(y) < N_i(x)$ must have been transmitted at least one unit earlier, and this is established by the inductive hypothesis.

Second, i must have transmitted the indication that level $h_i(x)-1$ is completed at least one unit earlier. For $\ell = h_i(x)-1$, the right hand side of (A2) is $n-2+h_i(x) + N_i'(\ell)$. Since all the nodes at level ℓ or less are transmitted before x, $N_i'(\ell) \leq N_i(x)-1$, so that by the induction hypothesis, i transmitted the completion of level ℓ in time. Third, i must have received a message that x is $h_i(x)-1$ hops from some neighbor j. Since $N_j(x) \leq N_i(x)$ by lemma 1 for the first such j, induction establishes that j transmitted x at least one unit earlier. Next consider (A2) for an arbitrary i,ℓ . Two conditions are necessary here for the time bound to be met. First, i must have earlier transmitted all x at level ℓ or less from i, which follows immediately by induction from (A1). Second, i must have received the induction that each neighbor has completed the list of nodes at level ℓ-1 (i.e. i must have not only transmitted all nodes at hop level ℓ, but must know that it has done so). From lemma 2, $N_j'(\ell-1) \leq N_i'(\ell)$ for each neighbor j, and thus by induction, $N_j'(\ell-1) \leq n-1 + (\ell-1) + N_j'(\ell-1)$, completing the proof.

Algorithm D1 in Pidgin Algol

It is assumed in the following algorithms that an underlying mechanism exists for queueing incoming messages to a node and for queueing outgoing messages until they can be transmitted on the outgoing edges. The local algorithm given for each node then indicates first the initial conditions and operations for the node and second the response taken to each incoming message. Each incoming message is processed in the order of arrival (or at least in order for a given edge).

Each node also maintains a set of local variables, which are now described for algorithm D1. The variable "level" indicates the number of hops to the destination on the current minimum hop path, "inedge" indicates the first edge on this path, "count" indicates the number of edges on which acknowledgements are awaited, and "m" indicates the number of nodes found in the current iteration for which the given node is on the minimum hop path. Finally there is a state associated with each outgoing edge. There are four possible states for an edge--"unused" means the edge is not currently in the minimum hop tree, "in" means the edge is the inedge, "active" and "inactive" mean that the edge is on the outgoing part of the minimum hop tree, with "inactive" meaning that no new nodes using that edge will be found at the current iteration.

Finally, for brevity, the destination node is considered to be split into two parts, one an ordinary node that follows the same local algorithm as the other nodes, and the other the "root" which co-ordinates the algorithm. These two parts are joined by a single conceptual edge, which does not count in the hop lengths of the other nodes to the destination.

```
                    Root Node Algorithm D1
Initialization
     begin  ℓ : = 0;
            send (test 0) on edge end
Response to (ack m) on edge
    begin if m = 0 then halt (comment: algorithm is complete);
          ℓ: = ℓ + 1 send (synch ℓ) on edge end
```

Ordinary nodes Algorithm D1

Initial Conditions

```
   level = $\infty$; for each adjacent edge e, S(e) = unused
Response to (test $\tilde{\ell}$) on edge $\tilde{e}$
   if  $\tilde{\ell} \geq$ level
      then send (ack 0) on edge $\tilde{e}$
      else begin level := $\tilde{\ell}$; inedge := $\tilde{e}$;
         S(e) := in; send (ack 1) on edge $\tilde{e}$ end
Response to (synch $\tilde{\ell}$) on edge $\tilde{e}$
    begin m := 0; count := 0;
         if  $\tilde{\ell}$ = level + 1
            then for each adjacent edge e $\neq$ $\tilde{e}$
                 begin count := count + 1;
                       send (test $\tilde{\ell}$) on edge e end
            else for each adjacent edge e such that S(e) = active
                 begin count := count + 1;
                       send (synch $\tilde{\ell}$) on edge e end;
         if count = 0 then send (ack 0) on inedge end
Response to (ack $\tilde{m}$) on edge $\tilde{e}$
    begin count := count - 1; m := m + $\tilde{m}$;
         if $\tilde{m}$ > 0 then S($\tilde{e}$) := active;
                else if S($\tilde{e}$) = active
                     then S($\tilde{e}$) := inactive;
         if count = 0 then send (ack m) on inedge end.
```

Algorithm D2 in Pidgin Algol

Algorithm D2 is basically the same as D1 with some extra variables and features. First, each node must keep track of when it is a synch node or presynch node, and it does this by a node state, which has the possible values Normal, Synch, and Presynch. Nodes also have a variable ℓ representing the current iteration level, and a variable k used by synch nodes for the last iteration of the group.

```
Root node - Algorithm D2
Initialization
  begin j := 0; ℓ := 0; send (test 0) end
Response to (ack $\tilde{\ell}$, $\tilde{m}$)
  if $\tilde{m} = 0$ then halt (comment: algorithm is complete)
     else if $\ell < n^{2x}$
        then begin $\ell := \ell + 1$; send (synch ℓ) end
             (comment: this sychronizes first group)
        else if $\tilde{\ell} = \ell$ then send (synch j)
             (comment: this initiates count of synch nodes)
          else begin j := ℓ;
               if $\tilde{m} < n^x$ then $\ell := \ell + \lceil n^{2x} \rceil$
                              else $\ell := \ell + \lceil n^x \rceil$;
               send (synch ℓ) end
```

Ordinary nodes - Algorithm D2

Initial Conditions

```
  level = $\infty$; State = Normal; count = 0
```

Response to (test $\tilde{\ell}$) on edge $\tilde{e}$

```
  if $\tilde{\ell}$ $\geq$ level
     then begin S($\tilde{e}$) := unused;
                send (ack $\tilde{\ell}$,0) on edge $\tilde{e}$ end
     else begin
          if  count > 0 then send (ack $\ell$,0) on inedge
             (comment: this acks outstanding old synch message);
          level := $\tilde{\ell}$; $\ell$ := $\tilde{\ell}$; inedge := $\tilde{e}$;
          S($\tilde{e}$) := in; count := 0;
          send (ack $\ell$,1) on inedge end
```

Response to (synch $\tilde{\ell}$) on edge $\tilde{e}$

```
  if $\tilde{e}$ $\neq$ inedge then send (ack $\tilde{\ell}$,0)
     (comment: this acks synch message on outdated tree branch)
     else if $\tilde{\ell}$ = $\ell$ + 1 and $\ell$ = level then execute procedure
                                                      xmit_test
          else if $\tilde{\ell}$ < $\tilde{\ell}$ and $\ell$ = level then execute procedure
                                                      count_synchs
             else if State = Synch then
                    begin k := $\tilde{\ell}$; execute procedure synchronize
                                                      end
                  else begin $\ell$ := $\tilde{\ell}$; execute procedure xmit_synch
                                                      end
```

<u>Response to ack $(\tilde{\ell},\tilde{m})$ on edge $\tilde{e}$</u>

```
if $\tilde{\ell} = \ell$ then   (comment: this ignores outdated acks)
begin if $\tilde{m} > 0$ then $S(\tilde{e})$ := active
            else if $S(\tilde{e})$ = active then $S(\tilde{e})$ := inactive;
      $m := m + \tilde{m}$; count := count -1;
      if count = 0 then
      begin if State = Normal then send (ack $\ell$,m) on inedge
             else if State = Synch then execute procedure synchronize
                 else if m > 1 then
                     begin m := 1; State := Synch; send (ack level, 1)
                                                          end
                     else begin State := Normal; send (ack level, 0)
                                                          end;
      end;
end
```

<u>Procedure xmit_test</u>

```
   begin m := 0; count := 0;  $\ell := \ell + 1$;
      for each edge e $\neq$ inedge do
          begin count := count + 1; S(e) := Unused;
               send (test  $\ell$) on edge e end
      if count = 0 then send (ack $\ell$,0) on inedge end
```

Procedure xmit_synch

```
begin m := 0; count := 0;
   for each edge e such that S(e) = active do
      begin count := count + 1; send (synch ℓ) on edge e end;
   if count = 0 then send (ack ℓ,0) on inedge
      (comment:  count must be non-zero for synch and presynch nodes)
                                                               end
```

Procedure count_synch

```
if m = 0 then send (ack level,0) on inedge
   else begin State := Presynch; ℓ := ℓ + 1;
              execute procedure xmit_synch end
```

Procedure synchronize

```
if m = 0 or ℓ = k
   then begin State := Normal; send (ack k,m) on inedge end
   else begin ℓ := ℓ + 1; execute procedure xmit_synch end
```

Algorithm for Destination to find number of nodes

We again assume that the destination is represented as a root connected to an ordinary node.

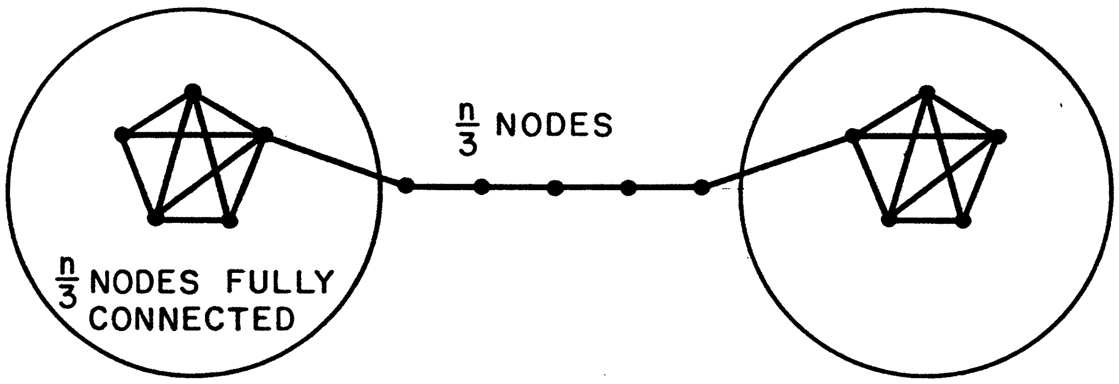
Root node

```
  Initialization
       send message
  Response to (ack $\tilde{n}$)
       halt (comment: $\tilde{n}$ is the number of nodes)
```

Ordinary nodes

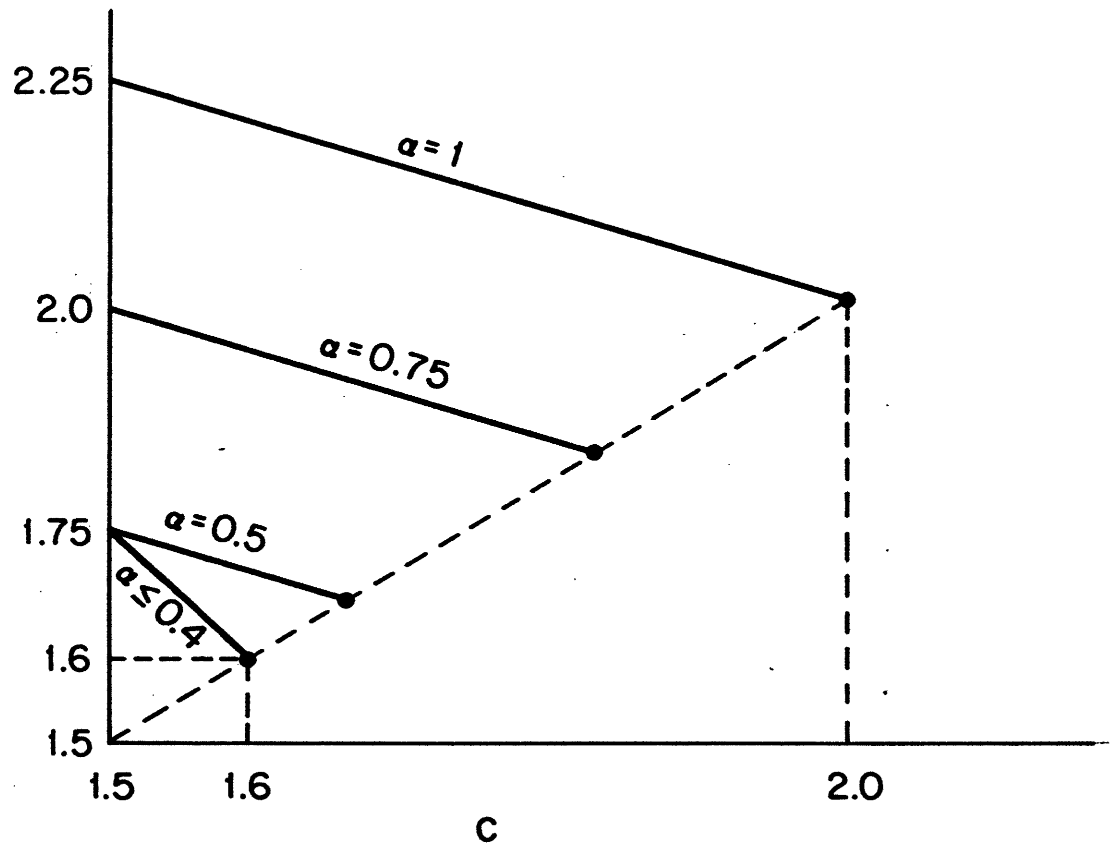
```
  Initial conditions
       n = 0; count = 0
  Response to message on edge $\tilde{e}$
     if n = 0 then
       begin n := 1; inedge := $\tilde{e}$;
             for each e $\neq$ inedge do
                  begin count := count + 1; send message on e end;
             if count = 0 then send (ack 1) on $\tilde{e}$ end
       else send (ack 0) on $\tilde{e}$

   Response to ack $\tilde{n}$ on edge $\tilde{e}$
     begin n := n + $\tilde{n}$;
          count := count -1;
          if count = 0 then send (ack n) on inedge end
```



DUMBELL NETWORK

Figure 1



COMPUTATION, TIME TRADEOFF

Figure 2