Efficient Algorithms for Elections in Meshes and Complete Networks*

July 1985

TR 140

Gary Lynn Peterson
Department of Computer Science
University of Rochester
Rochester, NY 14627

Abstract

The problem of electing a leader in distributed networks of processors where the topology is fixed and known is studied. In the case of an n node complete network, an algorithm with message complexity $O(n \log n)$ and message delay time $O(n)$ is given. These bounds are optimal to within constant factors. Also studied are meshes of processors. In the case of a square bidirectional mesh, an algorithm is given that has message complexity $O(n)$ and message delay time $O(\sqrt{n})$, which are again optimal. The mesh algorithm can be generalized to give efficient algorithms for unidirectional and rectangular meshes. Both sets of algorithms use the same fundamental idea to achieve their efficiency, indicating that the technique is both powerful and general.

*This research was supported in part by NSF grant No. MCS-8202950

1. Introduction

The *election problem* in a network of processes is to elect a process from an initially symmetrical set of processes. The processes are uniquely identified by a value from a set in a total order. No process has any global knowledge of the network, not even the number of processes, except perhaps for the topology. The process are to communicate via messages, using their identifier values, to uniquely elect a process. A formal definition of the model and the problem is given by Burns [Bur]. This problem is of general interest since many network algorithms begin with an election so that the elected process can be in charge of the computation. (Hopefully, the purpose is not to achieve a serialization of the set of processes, but to ensure a global state of control.) Hence it is important to better understand the inherent complexity of election in networks of processes.

There are several important measures for the complexity of concurrent algorithms in networks of processes. The one most often used as the primary measure is the total *number of messages* sent in the worst case in any execution of the algorithm. This measure primarily reflects the total load on the communication system of the network. While this is certainly an important measure, it perhaps is not as important as a suitable time measure. Since the networks to be discussed in this paper are asynchronous, the most natural time measure is that of *message delay time.* It is an analog to the time measures developed and used in [Pet1, P&F, Lyn]. An upper bound of one time unit is made on all message communications, local operations are not counted. The message delay time complexity is then the maximum over all possible execuctions within the limit on maximum message delay. This measure is not to be confused with any synchronous time measure such as that in [N&S]. While approriate in a synchronous setting, it can lead to peculiar results when applied to asynchronous algorithms. For example, an algorithm might have deadlock, and therefore in principle infinite running time, but the deadlock does not occur in the synchronous case, hence the running time would appear to be finite. The third measure to be used in this paper is that of *message size,* the maximum number of bits necessary to represent any message. Note that in some situtations a trade-off can be made between number of messages and message size. By encoding more information into longer messages, fewer messages can sometimes be sent. It is not clear, in general, if such a trade-off is actually beneficial in a particular use of an algorithm, so it is important to try to mimize both complexity measures.

The algorithms all assume that the election process starts simultaneously at all nodes. The algorithms are trivially modifiable to handle the case of only a subset of the nodes starting, and then at different times. The changes do not affect number of messages or message size at all. The time measure must be changed, however, so that only the worst case time from a node starting to finishing (i.e., being elected or ruled out of further contention) is counted. This is consistent with the measures use in [Pet1, P&F, Lyn].

The main results of this paper are made more interesting when compared to a result by Gallager which gives an upper bound of $O(n \log n + e)$ messages on election in an arbitrary network of n nodes and e edges [Gal, GHS]. It is natural to ask if this is the best one can do, not just in general, but also for particular networks. It is known, for example, that $\Theta(n \log n)$ messages are required in a ring network [Fra, H&S, Bur, Pet2, DKR, PKR.] Since a ring has n edges, it is clear that the larger term of the Gallager bound dominates the lower bound. Similarly, it is trivial to find many networks where the number of messages is $O(e)$ with $e = \Omega(n \log n)$, so that it appears that again the larger term dominates. It is natural to suppose that the lower bound for fixed networks is therefore $\Omega(n \log n + e)$, however, the two networks investigated in this paper show that this is not the case. In the case of complete networks, the number of messages is shown to be $O(n \log n)$ even though the number of edges is proportional to n^2, while in the case of square meshes, the number of messages is $O(n)$ $(=O(e))$. Hence, for these two networks, it is the smaller term in the Gallager bound that dominates. This implies that there is no simple general bound on the number messages for all fixed networks.

The election problem in a complete network of processes has been previously studied by Korach, Moran and Zaks [KMZ] and by Matsushita [Mat]. Korach, Moran and Zaks give an $\Omega(n \log n)$ lower bound on the number of messages to perform an election in a complete graph. They also present an algorithm for solving the problem that uses $O(n \log n)$ messages, however, the algorithm is extremely long and complicated and it is very difficult to determine its correctness. In addition, they do not analyze the message delay time of their algorithm. Matsushita studies a slightly different version of the problem and comes up with an $O(n)$ message algorithm. A better form of the same basic algorithm appears in [LMW]. The difference between the two version lies in the assumptions made about the knowledge of the connectivity. In [KMZ], no knowledge is assumed, other than the fact that it is a complete graph, so that the labels of edges out of a node contain no useful information. In [Mat], on the other hand, the edge labels do contain a small amount of topological information which can be exploited to give the much better bound. In an arbitrary network, such information may not be suitable, but if it can be assumed, then it is clear that better algorithms result. In this paper, the worst case is assumed. See also Santoro's paper [San] for a further discussion of topological awareness. The algorithm for complete graphs is remarkably simple, especially compared to the algorithm in [KMZ], and it is also efficient: the number of messages sent is $O(n \log n)$, the message delay time is $O(n)$ and message size is nearly optimal. More recently, Afek and Gafni [A&G] give another simple and efficient algorithm for this problem.

Matsushita also gives an algorithm for election in a square, wrap-around mesh of processes. Her algorithm is based on the one in [Pet2] and uses $O(n \log n)$ messages. The algorithm to be given here uses just $O(n)$ messages (which is clearly optimal up to constant factors) and also has message delay time $O(\sqrt{n})$ (again optimal up to constant factors). The message size is not as good in the complete graph case, but is still within a constant factor of optimal. The algorithm can be modified to work efficiently in a mesh where unidirectional communication alone is allowed. Also, if the mesh is not square but rectangular with length l and width w ($l \geq w$), then the algorithm can be adapted to use $O(n + l \log l/w)$ messages.

The same assumptions are made concerning the initial state of the processes before the election and what happens once a process has determined it is elected as in [Pet2]. Note that not necessarily the largest numbered process gets elected in either algorithm. If the largest (or smallest) is desired, it is a trivial matter for the elected process to find it and switch control.

The operations of sending and receiving messages will be represented in the algorithms by the operations **send**(*type*, $field_1$, $field_2$, ... ; *edge*) and **receive**(*type*, $field_1$, $field_2$, ... ; *edge*), which mean (respectively) send a message of type *type* with fields $field_1$, $field_2$, ... (any of which may be **nil**) along edge *e*, and wait for a message to be received, putting the values of the type, fields, and edge in the specified variables. If any component of a **receive** is a fixed value, then the process waits until a message satisfying those properties arrives and it is the next message processed, all others are queued. Otherwise, it is assumed that messages sent from one process to another are received in the order sent.

2. Algorithm for Complete Graphs

The algorithm for complete graphs has much in common with the algorithms in [Pet2]. All processes are assumed to be initially active and all but one become relays as the computation proceeds. Active processes operate in phases. The idea is to have the number of active processes be cut in half on each phase, giving at most $\log n$ phases with a linear number of messages per phase for a total of $O(n \log n)$ messages. As in [Pet2] the basic rule is to go on to the next phase if and only if you see an identifier value (*id*) that is smaller that your own and you are seen by a process with an *id* smaller than your own. Getting seen by another process and seeing another process is a remarkably simple and efficient process.

In order to see another process, every active process sends out a message of type "Looking" to some neighbor. If that neighbor is also active, then an acknowledgement message of type "Saw" is sent back with the *id* of the neighbor. Hence the first process has seen another process and waits to be seen and the second has been seen by at least one process. However, the neighbor may be a relay process. If the relay process has not previously received a "Looking" message on that phase it tells the sender that it is a relay but saves the *id*. Then later if another active process sends a "Looking" message on that phase, the relay can tell the other process that it has seen the first process and the first process that is was seen by the later one. Therefore every active process continues to send out "Looking" messages until it gets a "Saw" message from either an active or relay process, while simultaneously waiting to be seen (hopefully) by at least one other process with a smaller *id*. If it does get seen by a smaller numbered process and sees a smaller numbered process then it goes on to the next phase. A process will only become a relay when it sees a message with a higher phase number (which must be a "Looking" message). Note that each active process will see exactly one other process. (Unless there are processes on a higher phase, in which case it doesn't matter). On the other hand, each active process may be seen by any number of others, including none. At the end the last active process will receive "Relay" messages from all of its neighbors and the election can be terminated.

The algorithm for election in complete graphs follows. Messages have three fields: a type field ("Looking", "Saw", "Seenby", and "Relay"), a phase count field, and an *id* field. Let myid denote the process's unique identifier and E is the set of names of its edges. The variable myphase will be the current phase count of the process. The variable untried will keep track of those edges along which "Looking" messages have not yet been sent on this phase. (Note that there is no need to keep track of which neighbors are relay processes between phases.) The seenbyid variable will be the *id* of the lowest numbered process that has seen the process this phase. The sawid variable will hold the *id* of the process that was last seen.

```
myphase := 1
repeat
begin /* a phase */
   sawid := seenbyid := myid /* initially haven't seen or been seen by someone smaller */
   untried := E /* edges not yet tried */
   Send: if untried = Ø then announce elected /* all others are relays */
   e := any(untried) /* pick an edge */
   untried := untried - e
   send("Looking",myphase,myid:e) /* look for another id */
   repeat
   begin
      receive(type,otherphase,otherid;e') /* wait for a message */
      if otherphase > myphase then goto Relay /* become a relay if someone at a higher phase */
      elsif otherphase = myphase then
         case type
            "Relay": goto Send /* if relay then try another edge */
            "Saw": sawid := otherid /* you saw someone else */
            "Seenby": seenbyid := min(seenbyid, otherid)  /* smallest that sees you */
            "Looking": /* you are seen directly by someone else */
               begin
                  seenbyid := min(seenbyid, otherid)
                  send("Saw",myphase,myid;e') /* acknowledge */
               end
   end
   until sawid < myid and seenbyid < myid /* rule for going on to next phase */
   myphase := myphase + 1
end
until false
```

```
/* only a "Looking" message from a higher phase brings you here */
Relay: send("Relay",otherphase,nil;e') /* respond to "Looking" message */
e := e' /* remember who and where it came from */
sawid := otherid
myphase := otherphase
repeat
begin
   receive(type,otherphase,otherid;e')
   if otherphase > myphase then goto Relay /* received message from a higher phase process */
   elseif otherphase = myphase then
      begin /* send "Seenby" to original sender and "Saw" (original) to later ones */
         send("Seenby",myphase,otherid;e)
         send("Saw",myphase,sawid;e')
      end
end
until false
```

The correctness of the algorithm follows from two observations. First, the number of active processes is at least halved during each phase. This is because each active process that survives a phase must have been seen by a smaller numbered process and that process cannot survive the phase. Therefore the algorithm must terminate as the number of active processes is decreasing on each phase. The second observation is that at least one active process survives a phase up until the last phase when there is only one process left. Consider the graph consisting of the active processes on a phase with edges induced by the relation "saw". Since every active process sees exactly one other process, unless there is already a process on a higher phase, the out-degree of all nodes in the graph is one. This implies there is at least one loop in the graph. Consider the largest numbered process on a loop. It must survive the phase since it has been seen by a smaller process and the process it saw is smaller.

The message complexity of the algorithm is $\Theta(n \log n)$. On each phase there is at most one "Relay" message per relay process and at most one each "Seenby" and "Saw" message per active process. Each "Relay" and "Saw" has a corresponding "Looking" message that initiated it. Hence the number of messages per phase is at most $2n + A_i$ where A_i is the number of active process on the i^{th} phase. Since $A_i \leq A_{i-1}/2$ and A_1 is n, the total number of messages is at most $2n \log n + O(n)$. It is clear that executions can be constructed which achieve this bound.

The full proof that the message delay time complexity of the algorithm is $\Theta(n)$ is left to the completed version of the paper. The basic idea is to prove that no process that survives the i^{th} phase can spend more than n/A_{i+1} time during the i^{th} phase "on average". Hence the sum over all phases of the last surviving process is $O(n)$. The whole argument is quite involved.

Each message is of length $b + \log \log n + O(1)$ where b is the length of the longest identifier (which must be at least $\log n$, of course). Hence this algorithm is nearly optimal in terms of the number of bits per message.

Some simple variations which improve the various complexity measures include: keeping track of the relay processes across phases (a very small improvement) and not responding to messages from a lower numbered process (they will become relays anyway). Also the message type "Seenby" can be eliminated by using "Looking" instead and testing which edge it came from to determine if a "Saw" response is necessary. None of these produce a really significant improvement.

3. Algorithm for Mesh

The algorithm for bidirectional mesh is also similar in some ways to the algorithms in [Pet2]. Again, there is the concept of active and relay proceses, although not as strictly kept as before. The number of active processes at each phase will be cut by a constant factor (but not in half). Since the goal is an algorithm with message complexity O(n), the number of messages sent per phase must decrease by a constant factor each time, else O(n log n) or similar behavior results.

The basic goal of each active process on the i^{th} phase is to mark off the boundary of a square distance d ($d = \alpha^i$) on a side. This is done by sending a message distance d to the right then d down, d left, and finally d up to the original starting point. (Distances include the starting point.) Looking at things from the message's point of view, we see that a message marking off the boundary of a square may encounter the previously marked off boundary of another square on the same phase. This is how the actions of seeing and being seen can be carried out. The actual α to be used is left until the analysis. It must be chosen so that the proportion of active processes that become relays is high but not so that the number of messages per phase is too large.

During the marking of a boundary, if a message encounters another processes's boundary, it notes the *id* (this will be the only other process that the sender will see). Another message gets generated which follows the other boundary to its starting point telling the other process that it has been seen by another process with the specified *id*. Hence, a process may see many other processes's *ids* but only remembers one, or it may not see any others at all. It may also be seen by any number of other processes. There are two rules for moving on to the next phase. The first is the same as before: a process sees and is seen by a lower numbered process. When a process sees no other process at all, it is also permitted to go on to the next phase. (It, after all, may be the only active process left.)

The last part of the algorithm is invoked when $d \geq \sqrt{n}$. In this case a process sees its own marking message come in from the left rather than from the bottom. At this point there are only a small number of active processes left and a naive method of election among the remaining ones will do. Here a simple method along the lines of Chang and Roberts [C&R] is used. Basically, once a process sees that it has wrapped-around, it then sends a message down. If that message encounters the boundary of another wrapped-around process, it proceeds only if it is larger and is halted otherwise. Since there is no *a priori* way to determine if the boundary is a complete wrap-around yet, the algorithm must handle the case of a process failing to have its message wrap-around, but in the meantime stopping the message of some other process in its final stage. If this happens, the second process will send a clearing message backwards which restarts any such messages. Clearly, the largest numbered process that survives to the final stage will be elected when it sees its own message returning from the opposite direction.

The algorithm for election in meshes follows. There are four message types: "Looking" (for marking off a boundary initially), "Saw" (for marking off a boundary after seeing a smaller numbered process), "Seenby" (for following an intersected boundary), and "Finale" (for the last stage of the algorithm). The three fields of messages hold the phase number, *id* and distance. The variables used are fairly similar to those in the previous algorithm. A simplification is made in marking and following boundaries so that these operations are done only when it is necessary, e.g., there is no point in continuing to mark once a larger numbered processes boundary is encountered. The directions of right, down, left, and up are assumed to be numbered 0, 1, 2, and 3 respectively.

```
myphase := 1
repeat
begin /* a phase */
  sawnone := false /* haven't yet seen or been seen by anyone */
  sawsmaller := false
  seenbysmaller := false
  sawid := myid /* you are part of your own boundary */
  dist := $\lceil \alpha ** \text{myphase} \rceil$ - 1 /* distance to travel along boundary */
  send("Looking",myphase,myid,dist;0)   /* start marking a boundary */
  repeat /* both active and relay always do the following */
  begin
    receive(type,otherphase,otherid,otherdist;dir) /* receive a message */
    if dir = 2 and otherid = myid then goto activefinale /* wraparound */
    if type = "Finale" then goto relayfinale /* end of first part of algorithm */
    dir := dir + 2 mod 4 /* direction to send it on */
    if otherdist = 0 then /* "turn a corner" */
      begin
        otherdist := $\lceil \alpha ** \text{otherphase} \rceil$
        dir := dir + 1 mod 4
      end
    if otherid = myid and otherphase = myphase then /* a message for an active process */
      if type = "Looking" then sawnone := true /* saw no other id */
      elseif type = "Saw" then sawsmaller := true /* saw a smaller id */
      else seenbysmaller := true /* seenby a smaller process */
    elseif otherphase > myphase and type ≠ "Seenby" then
      begin /* become part of the boundary of the other process */
        send(type,otherphase,otherid,otherdist-1;dir) /* relay it on */
        myphase := otherphase /* remember pertinent details */
        sawid := otherid
        sawdist := otherdist
        sawdir := dir
      end
    elseif otherphase = myphase then
      if type = "Looking" then /* intersecting boundaries */
        if  otherid > sawid then
          send("Saw",otherphase,otherid,otherdist-1;dir) /* second sees first smaller */
        elseif sawid ≠ myid then
           send("Seenby",myphase,sawid,sawdist-1;sawdir) /* first sees second smaller */
          else seenbysmaller := true /* active receives another's "Looking" */
      else send(type,otherphase,otherid,otherdist-1;dir) /* relay it on */
  end
  until sawnone or sawsmaller and seenbysmaller /* rule for going on to next phase */
  myphase := myphase + 1
end
until false
```

```
relayfinale: saveid := myid
 if otherphase = myphase then goto test
else goto pass
activefinale: send("Finale",myphase,myid,nil;1) /*start final stage */
repeat
begin
  receive(type,otherphase,otherid,nil;dir) /*wait for message */
  if otherphase ≥ myphase then
    if type ≠ "Finale" then send("Finale",otherphase,otherid,nil;2) /*send clearing message back */
  else
  begin
    test: if dir = 0 then /* a clearing message from right */
      begin
        if saveid ≠ myid then send("Finale",myphase,sawid,nil;1) /* restart a message */
        if otherid ≠ myid then send("Finale",myphase,otherid,nil;2) /* pass it on */
      end
    else
    begin
      if otherid = myid then announce elected /* message has wrapped-around the other way */
      if otherid > sawid then
      pass: send("Finale",otherphase,otherid,nil;1) /* pass on if larger */
      else saveid := otherid /* save id for restart */
    end
  end
end
until false
```

The correctness of the algorithm follows from essentially the same principles as the previous one. That at least one process survives a phase is easy since either there is at least one process that doesn't see another, and that one moves on, or they all see just one other and the complete graph algorithm argument holds. The number of active processes per phase does decrease overall, but not as before. For example, it is possible for just one process to survive the second phase and then that process spends several phases just increasing its boundary. However, it is possible to bound the maximum number of active processes on each phase, and that bound strictly decreases. Let A_i be the maximum number of active processes on the i^{th} phase. (Clearly one need not worry about too many processes since any extra processes can delay themselves an arbitrary amount of time.) The bound on A_{i+1} has two parts: n/d^2 which is maximum number that can go on because they saw no other process, and $(A^i - n/d^2)/2$ which is at most half the remaining (for each that survives the phase by the other rule, at least one other process doesn't). Hence $A_{i+1} \leq (A_i + n/d^2)/2 \leq n/\alpha^{2i}(2-\alpha^2)$. Since d is increasing by a constant factor $(=\alpha^2)$ each phase, the upper bound on the number of active processes is decreasing. In the final part of the algorithm (when d exceeds $\sqrt{n}$), the number of processes surviving is at most one per horizontal line, but it is not too hard to prove that only a constant number (depending on α) survive to that stage. It is clear that only the largest numbered process remaining at that stage will go on and become elected.

Since the number of active process on the i^{th} phase is bounded by $n/\alpha^{2i-2}(2-\alpha^2)$ and each active process causes at most $8\alpha^i$ messages per phase (a factor of 4 to mark its boundary and another factor of 4 to follow the boundary of any process it sees), the total number of messages sent over all phases (excluding the final part of the algorithm) is $O(n\alpha^2/(2-\alpha^2)(\alpha-1))$. Optimization gives $\alpha \approx 1.1795$. The number of messages sent in the final part is $O(\sqrt{n})$ since at most a constant number survive to that stage. Therefore the total number of messages sent is $O(n)$.

The preceding analysis ignores the fact that α^i is not going to be integral. Clearly the distance

to travel must be rounded up and this has to taken into account in the analysis. However, it is not a large effect since the rounding amounts to at most 8 extra messages per process per phase with an insignificant change in the bound on the number of processes that are active on each phase. Hence this adds just a small amount to the constant factor.

The message delay time complexity of the algorithm is $O(\sqrt{n})$. The final stage of the algorithm is clearly $O(\sqrt{n})$. In the main part of the algorithm, the message delay per phase is $O(\alpha^i)$ and therefore $O(\sqrt{n})$ total.

The size of each message is considerably bigger than in the complete graph case: $b + \log n + O(1)$. Since b may be as little as $\log n$ this means that each message may be as much as twice as long as necessary. It seems plausible that a bound must be placed on how far a message may travel so that the number of messages doesn't get too large at the last stages of any type of algorithm, but such conjectures are notoriously prone to being proven false.

4. Variations of the Mesh Algorithm

In the case of a unidirectional mesh, where messages can only be sent right and down, the marking of a square must be done differently. Two messages are sent out: one d right then d down, the other d down and d right. They "rendevouz" at a point on the opposite side of the boundary from the initiating process. The process at that location then takes over for the original process. When either message first encounters another boundary on the same phase it sends a "Seenby" type message along that boundary. Note that each process may see either none, one or two other processes. The rules for advancement are now: if you see no other process or if you see only processes smaller and are seen by at least one smaller process. The final stage of the algorithm can be similarly adapted.

The points of correctness and complexity are slightly changed. That at least one process surives each phase follows from considering again the induced "seeing" graph, but only taking the largest process that each active process sees. The argument is then the same as before. Since an active process advancing by the second rule is seen by a smaller process which may also be the smaller process that sees some other advancing process, as many as 2/3 of the processes that saw another process may advance. Hence $A_{i+1} \leq (2A_i + n/d^2)3 \leq n/\alpha^{2i}(3\text{-}2\alpha^2)$. The total number of messages is therefore $O(n\alpha^2/(3\text{-}2\alpha^2)(1\text{-}\alpha))$ which is optimized at $\alpha \approx 1.10369$. The time and message size arguments are unchanged and therefore the bounds are the same.

In the case of a rectangular mesh of length l and width w ($l > w$, but neither not necessarily known), the algorithm can be very simply adapted to give an algorithm whose message complexity is $O(n + l \log l/w)$. The first part of the algorithm needs to be changed very little and the number of messages for that part is still $O(n)$. In the last stage, a more efficient method along the lines given in [Mat] (based on the algorithm in [Pet2]) is necessary. After the last remaining processes have wrapped-around, then they perform an election in the long (other) direction. There are $O(l/w)$ processes remaining and messages in each phase will add up to l total, so the result follows. Note that the mesh must be very thin, almost a ring, for the second term to come to dominate. In fact if $l = n$ and $w = 1$ then the result is equivalent to a ring and an optimal algorithm results. This adaptation can also be combined with the unidirectional one.

The mesh algorithms are obviously much more complicated than the algorithms for rings and complete networks. In addition, they are not as efficient as one would like. The original version of the bidirectional algorithm was somewhat simpler and more efficient, but it has two problems. First. its message delay time is $O(n \log n)$ and reducing that eliminates the gains in simplicity and efficiency. Second, it could not be adapted for unidirectional meshes. It is not known if there exists an algorithm for both kinds of meshes that is efficient in all measures and is also simple.

It is a very simple matter to extend the algorithm to higher dimensions. Just perform an efficient election in two dimensions, then a naive algorithm performed in each other dimension will work well from that point. Since the first part uses only a linear number of messags but considerably reduces the number of processes, the second part turns out to be efficient also.

One assumption that the algorithm uses that may be questioned is the assumption that the directions are known to all processes, i.e., the edges are labelled. This is unlike the complete graph case. The algorithm can be modified however so that no directional information is needed. First note that the algorithm only needs to mark off a square, the orientation of the square is irrelavant. Hence the algorithm just needs to simulate two kinds of message relaying: pass the message on in the opposite direction and make an appropriate turn. Both can be done in either the bidirectional or unidirectional case. Consider the later. A node has two incoming edges and two outgoing edges. When a message comes in, if the distance count is not zero, it is to pass the message along the edge corresponding to the incoming edge, else it sends it out the other edge. However there is no way a node can locally tell which edges correspond. But by doing a linear message, constant time preprocessing phase first, the nodes on the outgong edges can tell if they are the intended recepients. Hence the node sends the message to both and each will determine if the message is meant for them with the incorrect one just discarding it. The number of messages is twice the original number plus the preprocessing phase. the preprocessing phase consists of every process labelling a message by *id* and outgoing edge number and sending it to both outgoing neighbors. They in turn relay them on, adding edge labels. Every node has now received a message from every other node distance two away. It will have gotten two messages from the node 1 up and 1 left away. The rest are ignored. It now knows that it is at right angles to that node and therefore which incoming labels correspond to which outgoing lables for the intervening nodes. Hence, if it receives a message from a node that is supposed to go straight, it checks to see if the edge label (attached to the message by the sender) corresponds to the edge label of the node opposite it along that edge determined in the preprocessing phase. If so, it passes it along, else it discards it. The bidirectional case is (not surprisingly) a little more complicated.

5. Open Problems

The only significant open probem concerning the two types of networks studied here is the issue of finding a simpler election algorithm for the unidirectional mesh. Unlike the situation with rings, the unidirectional version appears to be significantly harder to solve. Further research on this problem is needed.

The next step to take in the study of elections in fixed networks is to find more examples of networks whose message complexity differs from the Gallager upper bound. There is no doubt that one can contrive networks whose message complexity is neither $O(e)$ or $O(n \log n)$ but somewhere in between. It would be much more interesting, however, if the networks were natural. Of more interest would be to find networks whose message complexity is $o(e)$ and $o(n \log n)$, or to prove that no such networks exist. Such a proof would go a long towards finding a general guide to determining the inherent message complexity of election in fixed networks.

Acknowledgement

I am grateful to Dan Hirschberg for taking the time to find errors in some early versions of the mesh algorithms

6. References

[A&G] Afek, K. and Gafni, E., "Simple and efficient distributed algorithms for election in complete networks," TR CSD-840042, Computer Science Dept., UCLA, Oct. 1984.

[Bur] Burns, J.E., "A formal model of message passing systems," TR 91, Dept. of Comp. Sci., Ind. U., May 1980.

[C&R] Chang, E. and Roberts, R., "An improved algorithm for decentralized extrema-finding in circular configurations of processors," *CACM 22*, **5**, 281-283, May 1979.

[DKR] Dolev, D., Klawe, M., and Rodeh, M., "An O(n log n) unidirectional distributed algorithm for extrema finding in circles," *J. of Algorithms 3*, 245-260, 1982.

[Fra] Franklin, W.R., "On an improved algorithm for decentralized extrema finding in circular configurations of processors," *CACM 25*, **5**, 336-337, May 1982.

[Gal] Gallager, R.G., "Finding a leader in a network with O(e) + O(n log n) messages," Unpublished M.I.T. Internal Memo.

[GHS] Gallager, R.G., Humblet, P.A., and Spira, P.M., "A distributed algorithm for minimum weight spanning trees," *ACM TOPLAS 1*, **5**, 66-77, Jan. 1983.

[H&S] Hirschberg, D.S., and Sinclair, J.B., "Decentralized extrema-finding in circular configurations of processors," *CACM 23*, **11**, 627-628, Nov. 1980.

[KMZ] Korach, E., Moran, S. and Zaks, S., "Tight lower and upper bounds for distributed algorithms for a complete network of processors," *Proc. 3rd ACM SIAGCT-SIGOPS Sym. on Principles of Dist. Comp.*, 199-207, Aug. 1984.

[LMW] Loui, M.C., Matsushita, T.A., and West, D.B., "Election in a complete network with a sense of direction," Coordinated Sci. Lab., U. of Illinois, Feb. 1985.

[Lyn] Lynch, N.A., "Fast allocation of nearby resources in a distributed system," *Proc. 12th ACM Sym. on Theory of Comp.*, 70-81, Apr. 1980.

[Mat] Matsushita, T.A., "Distributed algorithms for selection," M.S. Thesis, Report T-127, Coordinated Sci. Lab., U. of Ill. Urbana-Champaign, July 1983.

[N&S] Nassimi, D. and Sahni, S.K., "An optimal routing algorithm for mesh-connected parallel computers," *JACM 27*, **1**, 6-29, Jan. 1980.

[PKR] Pachl, J., Korach, E. and Rotem, D., "Lower bounds for distributed maximum-finding algorithms," *JACM 31*, **4**, 905-918, Oct. 1984.

[Pet1] Peterson, G.L., "Concurrency and complexity," TR 59, Dept. of Comp. Sci., U. of Rochester, Aug. 1979.

[Pet2] Peterson, G.L., "An O(n log n) unidirectional algorithm for the circular extrema problem," *ACM TOPLAS 4*, **4**, 758-762, Oct. 1982.

[P&F] Peterson, G.L., and Fischer, M.J., "Economical solutions for the critical section problem in a distributed system," *Proc. 9th ACM Sym. on Theory of Comp.*, 91-97, May 1977.

[San] Santoro, N., "Sense of direction, topological awareness, and communication complexity," *SIGACT NEWS 2*, **16**, 50-56, Summer 1984.