Fault-tolerant networks and graph connectivity

Edward T. Ordman
Department of Mathematical Sciences
Memphis State University
Memphis, TN 38152

Abstract. Let the vertices of a graph denote computer processes which communicate by passing messages along edges. It has been a standard Computer Science problem to provide algorithms that let the processes solve problems jointly (e.g. leader election, clock synchronization). What if some of the processes are maliciously faulty, i.e. send messages calculated to sabotage joint algorithms? Here we review a few "byzantine agreement" algorithms with interesting graph-theoretic features and raise questions about graph connectivity and diameter (with a few answers).

Partially supported by National Science Foundation Grant Number DCR-8503922

1. Introduction.

In my preceeding paper on Dining Philosophers and Graph Covering Problems, I spent a little while discussing some computer problems as a jumping off point to discuss topics that most computer scientists would identify mainly as graph theory, questions that are of known interest to researchers in graph theory and combinatorics. In this paper I mainly discuss computer algorithms. I'll need some graph arguments in the proofs, but my impression is that the graph theory questions suggested here haven't been studied as widely as the questions mentioned in that paper. I hope the context I suggest will help make these questions seem more important, because I think the answers might have some practical applications.

There is a great deal of interest in fault tolerant algorithms -- systems or programs that function even when part of the system breaks down. For instance, if we have a

JCMCC 1(1987), pp. 191–205

distributed system -- a bunch of computers communicating over communications lines -- either some of the computers might malfunction, or some of the communications lines might garble messages, or other things might go wrong. In particular, I've become interested in networks where one or more processors go crazy in malicious ways -- they actively try to sabotage the joint process. Algorithms designed to protect against such failures are called "byzantine algorithms". A good place to find articles about such systems, for example, are the Proceedings of the annual SIGACT conferences on Principles of Distributed Computing. A large part of my work on this was motivated by [TPS] in the 1985 PODC.

The standard example of such a problem is the "Byzantine Generals" problem [P]. Suppose we have n processes, each communicating with all others. At time 0 one of them -- A in Figure 1 -- is supposed to send a message, either 0 or 1, to the others. Within a known length of time we want them to agree on the message. If all the processes are correct, it is easy; we might for security have B and C each send the other the message "A sent 1", for example. But what if A is faulty, and sends a

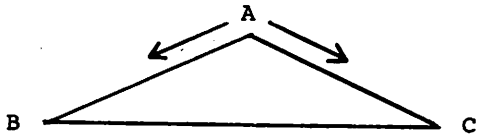


Figure 1.

1 to B and a 0 to C? B gets a message from C saying "A sent 0" -- but now how does B know if A is a liar, sending different messages, or if A really sent two 1's and C is a liar? Is there any way B can tell the difference?

Somewhat more formally, we want a program that we can load into computers A, B and C such that the programs will all stop at a known time after the "general" sends a message, have decided on a message, and we will have

(A) Correctness. If the general is correct, all correct processes will decide on the message sent by

the general;

(B) Agreement. Whether or not the general is correct, all correct processes must agree on the same message.

The original application was for control of an airplane's flaps; one wants them to obey the autopilot if it is working, but act in a self-consistent fashion even if the autopilot stops working or communications from it are garbled.

Here is a nice proof we cannot do this for three processes, one of which is faulty. This argument is due to [F]. Suppose my friend Joe appears in the door, with a computer program in hand, and he claims that if I load this program into computers A, B, and C, and start them running, the program will stop with a correct result. I am going to prove he is wrong, by the following strategy. First, I build a set of 6 processes, as shown in Figure 2. I load

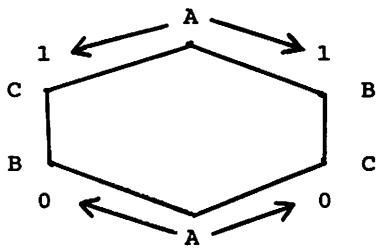


Figure 2.

his program into each of them, telling both processes labelled A that they are process A, and so on around. I let the top A be a general sending 1 and the bottom A be a general sending 0. Now the top A and B are both correct processes; they will act just like a correct A and B in a triangle with a faulty C. Since the general, A, is correct and sent 1, B must decide 1 by the Correctness property if Joe's program is correct. Now the right-hand B and C are correct, and act just like they were in a triangle with a faulty A; since B decides 1, C must decide 1 by the Agreement property. But of course exactly the same argument on the bottom A and C shows C must decide 0, a contradiction. Joe's program can't work; to sabotage it in

the triangle, the faulty A in the triangle just sends the B the same messages as the top A in the hexagon, and sends the C the same messages as the bottom A in the hexagon. The theorem one gets from this is: You can't hope to have byzantine agreement unless the correct processors outnumber the faulty ones by MORE than two to one. Similarly, it turns out that to do this sort of thing in a graph with f faulty processes, the graph must have at least 3f+1 processes and be at least 2f+1 - connected. The fact that you need at least 2f+1 - connectedness is very closely tied to Menger's Theorem. To be sure to get a message from A to B, I need to send it by 2f+1 vertex-disjoint routes (Figure 3). That way, even if f routes pass through faulty processes that change my YES to a NO, B will still receive a majority of YES messages, and know that at least one of those YESes really came from me (the faulty processes could have started f of them by themselves).

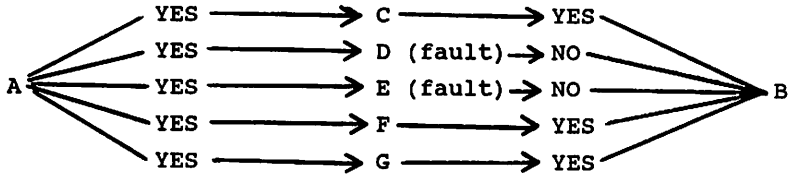


Figure 3.

There are a lot of papers out on some of the things one might want to do in a network, and many of these use "byzantine generals" algorithms as building blocks. For example, there are several problems associated with coordinating clocks of computers in a network: one may have a synchronous system where all clocks advance at the same moment, and we want them all exactly alike [NH], [BL]; or one may have an asynchronous system, where the real clocks run at slightly different speeds and we need to agree on an approximately accurate "logical time" that is similar for all processes[ST]. The first problem, incidentally, is called the byzantine firing squad problem.

2. Graph recognition, part 1.

Much of the published literature assumes that all

processes have direct communication with all others -- that is, the underlying graph is a clique. Obviously, if we are looking to use graph theory, we want to look at networks where not all processes are connected directly to all others.

What problem shall we work on? A good starting point seems to me to be to identify the graph we are in. Suppose each process has a unique identifier (ID), and knows how many communication lines it has. I'm going to assume all communications channels are two-way, so I have graphs, not digraphs. Can the processes find out what the network that they are in looks like? Let's start by supposing that all processes are correct and all messages are transmitted correctly, to get a notion of the problem (Figure 4).

Throughout this talk, I'm going to assume a synchronous environment, in the following sense: every process has a clock that starts at 0. In each time unit or round, each process can (1) receive any number of messages from adjoining processes; (2) compute; and (3) send any number of messages to adjoining processes.

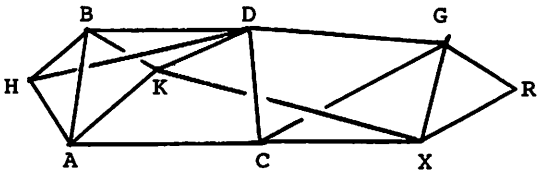


Figure 4.

The algorithm we will consider is given below. Clearly, we have no hope unless G is connected; so G has a finite diameter d. In this case, every process will have received the adjacency list of every other process by the end of round d; once it has heard from all processes the stopping condition in (2) will be satisfied and it will have a complete map of the graph. The number of messages sent is clearly polynomial since each of the n vertices forwards each of n adjacency lists (of length less than n)

once each to all or a subset of its neighbors (so less than n times).

A GRAPH RECOGNITION ALGORITHM

In round 0, each process sends its ID to its neighbors.

In round 1, each process receives its neighbors IDs and thus has its own adjacency list. It sends its adjacency list to all its neighbors.

In each subsequent round, each process

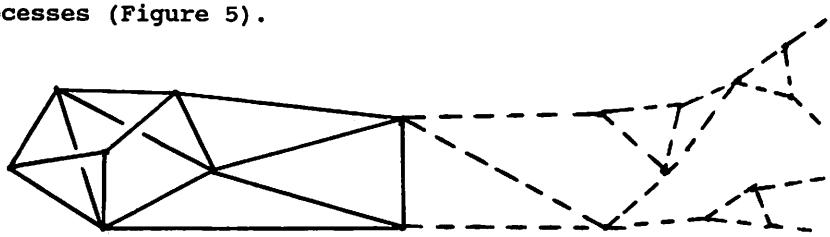
1) receives some adjacency lists;
2) makes a list of each process which appears in any adjacency list that has been received but whose own adjacency list has not yet been received. (If this list is empty, it can stop at the end of this round);
3) forwards each adjacency list it has received for the first time to every neighbor it has not received it from.

END of GRAPH RECOGNITION ALGORITHM

---

3. Graph recognition, part 2.

The above algorithm fails badly to extend to the byzantine case. A simple problem is that a faulty process may never send us its adjacency list; a much more serious difficulty is that each faulty process can invent fictitious processes connected to the rest of the graph through it; a correct process may continue indefinitely to receive adjacency lists from ever more distant fictitious processes (Figure 5).



Correct processes | Faulty processes | Fictitious processes

Figure 5.

Once we can put an upper bound the size of the graph

there is hope; we can use a byzantine generals algorithm, for example, to decide on the individual adjacency lists. The hard part is knowing when to stop listening and estimate the size of the graph. The key to this is the following: as long as there is a correct process we haven't heard from, there are f+1 correct paths from us to it; we are getting messages forwarded along f+1 disjoint paths. While a faulty process can send us a false adjacency list by f+1 or more paths, it cannot cause an adjacency list from a fictitious process to reach us by more than f disjoint paths, since each such path must contain a faulty process.

We need to be more explicit about our means of transmitting messages. From now on, each process sending a message will sign it; and each process forwarding a message it has received will append its signature to the end, so we can tell the route a message has followed. We assume that faulty processes can make up any nonsense they want, and forge any signature they want, with one exception: we will suppose each process knows who is on the end of its wires, so that it knows who it physically received each message from. Hence, a faulty process can't lie about its own name to its own neighbors: and a correct process will ignore any message that isn't signed by the person it receives it from.

Finally, we see that a message is credible (that is, we believe the sender sent it) only if (1) we sent it ourself, or (2) it was originated by an immediate neighbor and sent to us directly, or (3) we receive it by at least f+1 vertex-disjoint routes. In this case, we say that we have heard the message.

We need the following simple graph-theoretical lemma.

Lemma 1. If G is a graph with n vertices and is 2f+1 - connected, between any two nonadjacent vertices there are 2f+1 (vertex-)disjoint paths of length not exceeding n-2f-1. Further, there is a graph for which this is attained.

Proof. Let a and b be nonadjacent vertices. By Menger's Theorem, there are 2f+1 paths between them. Let one of these be as long as possible. The other 2f paths each use at least one vertex of G, and a and b occupy vertices. Thus at most n-2f-2 vertices are available to be interior vertices of the long path, and it consists of at most n-2f-1 edges. To see that this can happen, join all points of a cycle with n-2f+1 points to all points of a complete graph on 2f-1 points. To construct 2f+1 disjoint paths between points that are two apart on the cycle, we have the following: the path of length 2 on the cycle; 2f-1 paths of length 2 via the complete graph; and the long path around the far side of the cycle, of length n-2f-1.

If I'm going to guarantee that f+1 copies of my message will reach the recipient, I've got to send it by 2f+1 disjoint paths. How long will it take until f+1 copies get there? In the worst case, as long as the longest of the 2f+1 paths. We call this number d', the wide diameter of G. We've just seen that d' is no larger than n-2f+1.

We can now state an algorithm to find the vertices of the graph G. The algorithm appears on the following page.

We need to show that the "candidate graph" produced in each correct process by this algorithm has all the vertices of G and only those of G; that all the correct processes thus agree on n = n' and on d"; and that all correct processes determine d" no later than round d". Notice that the adjacency lists "heard" may differ from process to process and may not for any process represent a reasonable graph; a faulty process may have more than one adjacency list "heard" and it is possible that p_i reports p_j is adjacent to it but p_j claims p_i is not adjacent to it.

Lemma 2. No correct process stops before it has heard a message from every correct process.

Proof. We show that at round t>0, if correct process p_i has not heard from correct process p_j, then p_i has H at

least f+1. Suppose p_i has not heard a message from p_j. Then there are at least f+1 disjoint paths from p_j to p_i passing through correct processes. On each of those paths there is a closest process to p_j from which p_i has heard a message; these are all distinct. (By round 1, p_i has heard from all the processes adjacent to it, hence from one on each of these paths). Thus there are at least f+1 vertices that have been heard from that report adjacency to vertices (p_j or something closer to p_i) not yet heard from, and H is at least f+1.

A BYZANTINE GRAPH SIZE ALGORITHM

```
For each process:  At round 0:
   Send to all neighbors my own adjacency list (the message
     ("p_i adjoins p_j, ..., p_k, signed p_i");
   Store a "candidate set"  of vertices and of adjacency
       lists,  each  of which may be marked "heard"  or
       "expected".  Initially  this set consists of  myself
       and  my own adjacency list, marked "heard".
At round t > 0:
     Forward all messages received, adding a signature.
     When a new adjacency list is "heard",  add that vertex
       and adjacency list to the candidate set.
     Compute  H  := the number of  vertices  whose  "heard"
       adjacency lists contain vertices not yet "heard".
     IF H > f THEN
       For any vertex previously marked "expected" but not
          yet "heard", mark it "heard" (it is faulty);
       For any vertex included in "heard" adjacency lists
            of at least f+1 vertices but not yet "heard",
            add it to the candidate set marked "expected";
     ELSE (If H < f+1)
          n' := number of "heard" vertices;
          d" := n' - 2f -1;
          IF t > d" THEN  EXIT (this algorithm) ENDIF;
     ENDIF.
          END OF BYZANTINE GRAPH SIZE ALGORITHM
```

Lemma 3. Every correct process hears a message from every other correct process by time d'.

Proof. Given correct processes p_i and p_j, they are adjacent or there are 2f+1 disjoint paths between them with the longest not exceeding length d'. Hence the message sent by p_i at the start of the algorithm reaches p_j by at least f+1 routes within that time.

Lemma 4. By time d' + 1 every correct process will have every vertex of G marked "heard" in its candidate graph.

Proof. By time d' it will have heard a message from every correct process. It will have heard an adjacency list from every correct process. But then for each faulty process, at least f+1 correct processes are adjacent to it; so if no message from that faulty process has been heard by time d' it will become "expected" no later than then and will be marked "heard" the following round.

Lemma 5. No process not in G will ever be marked "heard" by a correct process.

Proof. A correct p_i may receive messages purporting to be originated by fictitious vertices, but each such message must have come from a faulty process and must therefore have a signature of a faulty process among its signatures; hence it can arrive by at most f disjoint routes and will not be "heard". A vertex marked "heard" by first being marked "expected" must be adjacent to f+1 "heard" processes, at least one of which must be a correct one; so it must really be in G.

Lemma 6. At all times after d', H will be f or less for every correct process.

Proof. By this time the candidate graph will have all correct nodes and all faulty nodes marked "heard". No message will have been heard from a correct node except a correct message (for no incorrect message can have arrived by f+1 disjoint routes) so at most f processes (the

incorrect ones) can have sent adjacency lists that have been "heard" and report not-yet-heard (hence fictitious) vertices.

Theorem 1. By time d' + 1 all correct processes will have a list of precisely the vertices of G; all will have the same value of n' equal to n, and all will have the same value of d" which is no less than d'.

Proof. By the lemmas, each correct process will have the same correct list of vertices, hence the same n' and d"; by Lemma 1, d" will be no less than d'.

4. Graph recognition, part 3.

The algorithm of Section 3 allows us to determine the number and names of the processes in the graph; it gives little help on the edges. This problem turns out to be much harder. One solution is to continue exchanging messages, carrying out a Byzantine Generals algorithm on each adjacency list. In [O] there is an algorithm that carries this out, terminating in 2(f+2)d" rounds -- surely not the best possible, but I suspect it is of the right degree. (A figure like (f+1)d' might be best possible). Once this is done, all correct processes will have the same candidate adjacency matrix, i.e. the same list of claimed adjacencies. Note that if A is faulty and B is not, A may claim to be adjacent to B but not conversely, or vice-versa. All correct processes might decide to suppose an edge exists if either end claims it; our graph image G' would now contain at least all edges of G that touch at least one correct vertex. In particular, the graph image G' would be sufficient to allow the processes to engage in further byzantine algorithms.

There is, however, a serious problem with G'. G is 2f+1 - connected; G' need not be (edges between two faulty processes in G need not appear in G'). Thus, there seems little possibility that we can compute any number very close to d'. G may have a very modest "wide diameter", that would allow quick byzantine agreement; G' may not let

us compute any wide diameter at all, forcing us to use the upper bound n - 2f - 1. If we have to wait that long to be sure a distant process has received our message, it will slow down byzantine algorithms seriously. In fact, the problem is worse: Since G is 2f+1 connected, we can actually determine 2f+1 paths between each pair of points: once we determine them (I'm afraid this is NP-complete) we can send addressed messages and the usual byzantine algorithms will require only polynomially many messages thereafter. Since we can't find 2f+1 paths for all pairs of points in G', we may have to resort to broadcasting messages on all possible paths -- hence, exponentially many messages!

There is a partial solution. Suppose the original graph G is at least 2.5f+1 - connected instead of just 2f+1 - connected. Now G has 2.5f + 1 paths between A and B, and if A and B are correct processes then at most f/2 of them contain edges that are absent in G' (an edge is absent only if both ends are faulty). Thus G' will have 2f+1 paths between correct processes, we can find a number that will function as the "wide diameter", and we can make do with polynomially many messages after our exponentially-nasty (in number of messages, and path finding) graph recognition algorithm.

This leaves some open questions that are heavy with graph theory. Given a graph G, is there a graph recognition algorithm that produces d' as a byproduct? Can d' in fact be determined? Can we determine some reasonable-sized collection of paths between each pair of points guaranteed to contain f+1 disjoint correct paths?

5. Path lengths in Menger's Theorem.

Finally we turn to some implications for pure graph theory. Let G be a graph with n vertices which is k-connected; Menger's Theorem tells us that any two vertices are connected by at least k vertex-disjoint paths. Lemma 1 gives us an upper bound on the lengths of these paths, and an example where the upper bound is assumed. But if I were

designing a network, I'd want to guarantee that there were lots of short paths. Hence, a question: what are some conditions on a graph that will guarantee the existence of a large number of short disjoint paths connecting any two points? Are any of these conditions easy to recognize, in a context such as the graph recognition problem above?

Some examples of "nice" graphs of this sort are discussed in [BHP]. A look at some related pathologies may be found by looking in [B], which discusses how badly the diameter of a graph may increase if a vertex is deleted. One theorem says, very approximately, that if G has even diameter D and maximum degree d, d>5, and s vertices are deleted, the diameter of the remaining graph cannot exceed sd(D-1) - 1. Loosely, that is, deleting a vertex increases the diameter by as much as dD. This seems rather frightening, but it just seems to show that one had better look at well-bahaved graphs!

In early 1986, R. Faudree, R. Schelp, and I at Memphis State had a chance to discuss these questions with Zsolt Tuza of Budapest. (The rough notes we have are cited as [FOST]). I'll try to suggest the way we are going. Suppose a graph G on n vertices is k-connected. Then any two points are connected by k vertex-disjoint paths. What conditions can we put on G to force all of these paths to have length less than some constant m? We have been looking at three kinds of conditions: degree, connectivity, and edge density.

Theorem. If G is a k-connected graph on n vertices and the degree of each vertex of G is at least 3n/s + k -1 then there are k vertex-disjoint paths of length not exceeding s between any two vertices of G.

Theorem. If G has connectivity at least n/s + k then there are k vertex-disjoint paths of length not exceeding s between any two vertices of G.

In each case there is an example which shows that at least the coefficient of n cannot be improved. For

example, consider a cycle of length s+2 and replace each point except 2 adjacent ones by a clique on (n-k)/s points, each edge by a complete connection between the corresponding cliques or points, and add in the center a clique on k-2 points attached to everything (Figure 6).

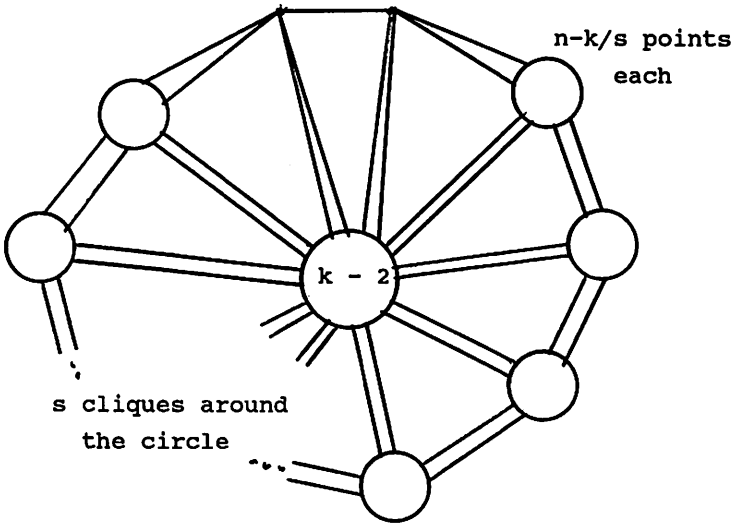


Figure 6.

This is (n-k)/s + (k-2) + 1 or (for large s) n/s + k - 1 connected; it has n vertices. The two single points are connected by one path of length 1, k-2 of length 2, but the k-th path has length s+1.

The situation for edge density harder to express, so I'll leave the details until we write up that paper. There are two graphs -- one essentially the above (motivated by the computer-motivated example I gave earlier) and the other made by deleting the edge between the two single vertices and adding another vertex in the center. We think that these two graphs -- well, these two families of graphs -- are extremal examples for graph density conditions on having k paths of length s.

I can't help but notice that the limiting cases in [FOST] seem to depend on the fact that adjacent vertices

require a long path. In our applications we only need many short paths between non-adjacent vertices. Does this change things? I don't know yet.

REFERENCES

[B] Bond, J., and C. Peyrat, Diameter vulnerability in graphs, in Y. Alavi et al, eds., Graph Theory with Applications to Algorithms and Computer Science, Wiley, New York, 1985, pp 123-149.

[BHL] Bermond, J-C., H. Homobono, and C. Peyrat, Large fault-tolerant interconnection networks, Proc. First Japan Internl. Conf. on Graph Theory and Applics., Hakone, Japan, June, 1986 (to appear).

[BL] Burns, J.E., and N.A. Lynch, The byzantine firing squad problem, MIT Lab. for Computer Science, April 1985.

[F] Fischer, M. J., N.A. Lynch and M. Merritt, Easy impossibility proofs for distributed consensus problems, Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Canada, August, 1985, 59-70.

[FOST] Faudree, R., E.T. Ordman, R. Schelp, and Z. Tuza, Menger's theorem for short paths, rough notes.

[NH] Nishitani, Y. and N. Honda, The firing squad synchronization problem for graphs, Theoretical Comp. Sci. 14(1981), 39-61.

[O] Ordman, E.T. Distributed graph recognition with malicious faults, Proceedings of First China-U.S.A. International Conf. on Graph Theory and its Applications, Jinan, China, June, 1986 (to appear).

[P] Pease, M., R. Shostok and L. Lamport, Reaching agreement in the presence of faults, J. Assn. Comp. Mach. 27(1980), 228-234.

[ST] Srikanth, T.K., and S. Toueg, Optimal clock synchronization, Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Canada, August, 1985, 71-86.

[TPS] Toueg, S., K.J. Perry and T.K. Srikanth, Fast distributed agreement, Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Canada, August, 1985, 87-101.