Abstract

System Level Fault Diagnosis under Static, Dynamic, and Distributed Models

William I. Hurwood
Yale University
1996

Consider a set of n processors that can communicate with each other. Assume that each processor can be either "good" or "faulty". We wish to *diagnose* the system. That is, we use tests between the processors to determine the status of each processor. We suppose that good processors are accurate, but that faulty processors may be in error. We develop fast parallel diagnosis algorithms, and also use adversary arguments to prove that our algorithms are near optimal. Our models are based upon the system diagnosis model proposed by Preparata, Metze and Chien [46].

We consider three different models of diagnosis. First we have a *static* model in which each processor has a fixed status, there is an upper bound t on the number of faulty processors, and we wish to minimize the number of rounds of testing used to perform diagnosis. We prove that 4 rounds are necessary and sufficient when $(8/3)\sqrt{n} \leq t \leq 0.03n$ (for n sufficiently large). Furthermore, at least 5 rounds are necessary when $t \geq 0.42n$ (for n sufficiently large), and 10 rounds are sufficient when

$t < 0.5n$ (for all n). It is well known that no general solution is possible when $t \geq 0.5n$.

Second we consider a *dynamic* model in which a processor may change status during the diagnosis. In each round up to t processors may break down, and we may direct that up to t processors are repaired. We show that it is possible to limit the number of faulty processors to $O(t \log t)$, even if the system is run indefinitely. We present an adversary which shows that this bound is optimal.

Third we consider several *distributed* models in which there is no centralized controller directing the tests, but instead each processor decides independently which tests it should perform. This problem is closely related to the collect problem [49]. We present a randomized algorithm that will, with high probability, complete diagnosis after $O(n \ln^3 n)$ tests. The best previous known algorithm needed $O(n^{3/2} \ln n)$ tests. It is clear that n tests are needed. We present an algorithm to perform dynamic distributed diagnosis. We use several competitive ratios to analyze the performance of this algorithm.

System Level Fault Diagnosis under Static, Dynamic, and Distributed Models

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

William I. Hurwood

December 1996

Contents

List of Figures

Acknowledgments

Throughout my studies at Yale, I have had the good fortune to have worked with an amazing and inspired group of individuals. Among these there are a few whom I would like to thank specially. First and foremost I thank my advisor, Professor Richard Beigel, for his exceptional insight and support. He introduced me to my subject of study and, always enthusiastic, would never decline an in depth discussion on difficult topics, inspiring new ideas and teaching me a great deal. I especially appreciate his support and concern in the writing of this thesis. His detailed proof reading, helpful comments, and exceptionally high standards were immensely beneficial.

Thanks also to Professor Lovász for his wonderful lectures on many areas of Combinatorics; to Professor Aspnes, who introduced me to the cooperative collect problem and who taught me how to make use of moment generating functions; to Professor Spielman, whose paper inspired my initial ideas; to Professor Seligman for supporting my decision to cohabit in the Computer Science department; to Professors Kahale, Westbrook, and Angluin for helpful discussions; and to Andrew Kotlov and Andy Beveridge, two of my colleagues in the Mathematics Department, with whom I had many enlightening discussions, who patiently listened to my talks, and who were of particular help to me.

I would also like to thank the people who motivated my interest in mathematics. First my Grandfather, who has always been ready to show me mathematical tricks, and who has continued to take a keen interest in my progress. Second, Mr. Sharples, my maths teacher at Eastbourne Sixth Form College who infected me with his enthusiasm for the subject. Also I would like to thank my fellow students, at school, sixth form and at Cambridge, especially James, Matthew, Michael, Marcus and Graham. Their presence at classroom, lectures and supervisions has always been challenging and a source of inspiration for me.

Finally I would like to thank those people who are close to me: my parents and my fiancée, Lee. Thanks for listening to my thoughts for all these years, and nodding at the right places.

Chapter 1

Introduction

Consider a computer system made up of thousands of processors. It is inevitable that some of the processors will fail. Fault-tolerant systems will function in the presence of a limited number of faults, but even these must be located and repaired before too many accumulate.

One approach to finding these faulty processors is by using some of the processors to test others. This is called system-level fault diagnosis. Each of n atomic processing units is assumed to be able to test every other unit. But when a faulty unit carries out a test the result is unreliable. Such testing protocols have been usefully implemented [17,16].

Preparata, Metze and Chien first proposed a fault diagnosis model in [46]. They suggested viewing the system as a graph with the processors as nodes and tests as edges. Each node is either *good* or *faulty*. We will refer to the original model as the PMC-model. Nakajima [44] introduced *adaptive* tests, where the tests performed in later rounds are chosen after considering the results of the earlier rounds.

This naturally led to asking how many rounds of adaptive tests are needed to carry out a complete diagnosis. This model was used in [30,32,12,31,13] in which the upper bound on the number of rounds of testing as a function of n was eventually reduced from a linear bound, through logarithmic and doubly-logarithmic bounds, to a large constant bound.

It is natural to remove the requirement that every processor have a fixed status throughout the testing procedure. Such *dynamic* models have been considered before [34,17,16]. In these papers the authors presented self-stabilizing distributed algorithms: if the processors stopped changing status then the remaining good processors would eventually discover the status of all the processors in the system.

Another topic of interest is diagnosis with repair. In *sequential diagnosis*, introduced in [46], the objective of the testing process is to discover one or more faulty processors, which can then be repaired before the testing process is repeated.

In this thesis we will present new and faster algorithms for three kinds of diagnosis: static diagnosis, dynamic diagnosis and distributed diagnosis. In each case we will prove a lower bound showing that our results are close to optimal.

1.1 Models and Results

1.1.1 Static diagnosis

Static diagnosis is the simplest model of fault diagnosis that we consider. There are n processors, each of which has a *fixed* status. There is a given bound, t, on the number of faulty processors. The system is diagnosed by performing tests between processors. Good processors are always accurate; faulty processors may be inaccurate. The testing proceeds in rounds in which each processor may participate, as tester or testee, in at most one test. The goal is to determine the status of each processor using as few rounds of testing as possible. The model is formally defined in Section 2.1.

This model has been studied extensively [44,30,32,12,31,13]. It is based on Preparata, Metze and Chien's seminal work [46], which introduced the model used for the system of processors, and for tests between the processors. Their model is *nonadaptive*. All the tests indicated by a fixed testing graph are performed. They were interested in *one-step* diagnosis, that is in characterizing graphs which provided sufficient information to perform complete diagnosis. The nonadaptive model was studied further in [29,22,52,12]. Nakajima [44] proposed the usage of adaptive tests.

Hakimi and Nakajima [30] showed that diagnosis could be completed in $n + 2t - 2$ tests, where each test is made adaptively. Blecher [18] improved this bound to $n + t - 1$ adaptive tests, and showed that this is the best result possible. These algorithms worked by identifying a good processor and then using it to diagnose the other processors in the system.

Hakimi and Schmeichel [32] observed that although $n + t - 1$ tests are needed, it is possible to speed up the diagnosis by performing tests in *parallel*. They introduced rounds of testing, in which each processor could participate in at most one test in each round. The diagnosis is obtained faster (i.e. after fewer rounds) but more tests are performed. They proved that diagnosis could be completed in $\lfloor \log_2 t \rfloor + 1$ rounds.

It is clear [46] that diagnosis cannot be performed unless $t < \lceil \frac{n}{2} \rceil$. So the most general problem in this model to to determine the maximum number of rounds needed to diagnose every processor, given only that there are n processors and that at most $t < \lceil \frac{n}{2} \rceil$ of the processors are faulty. We refer to this problem as the *general* diagnosis problem.

It is clear that the general diagnosis problem can be solved with n rounds of testing. Each processor tests every other processor. We view each diagnosis of a particular processor as a vote on its status. Then the status which receives the most votes must be the true status of the processor, because the good processors are in the majority.

Several useful techniques have been developed for fault diagnosis in the parallel model. Central to all these techniques is the construction of equivalence classes that consist either entirely of good processors or entirely of faulty processors. Schmeichel, Hakimi, Otsuka, and Sullivan [31] use a doubling technique to grow large equivalence classes; their algorithm uses $O(1 + \log_{\lfloor n/t \rfloor} t)$ rounds and $O(n)$ tests. Beigel, Kosaraju, and Sullivan [12] use a squaring technique to grow them in $O(\log \log n)$ rounds. In another algorithm, they use expander graphs to obtain large equivalence classes in

$O(1)$ rounds. Beigel, Margulis, and Spielman [13] use a special class of random graphs to obtain large equivalence classes, and complete diagnosis in 32 rounds (for sufficiently large n), and expander graphs to do so in 84 rounds (for infinitely many n).

We show how to solve the general diagnosis problem in 10 rounds. Unlike [13] our result holds for every value of n. The technique used is to perform tests in the first 6 rounds which build equivalence classes from some of the nodes. Taking advantage of the nonadaptiveness of the algorithm from [13] we diagnose the equivalence classes in 2 more rounds. Finally we improve on a technique from [12] to complete diagnosis in another 2 rounds. Our algorithm is more than three times as fast as the best previous algorithm for this problem.

Our proof, like that of [13], uses the probabilistic method [51]. Part of the proof is nonconstructive. We also give a deterministic polynomial-time algorithm, which relies on some preprocessing, that can solve the problem in 12 rounds.

Prior to this work the best known lower bound was 3 rounds [12]. We raise the lower bound for the general diagnosis problem to 5 rounds. So we have closed the gap from 3–32 to 5–10. The lower bound holds for all sufficiently large n, and for $t \geq \frac{5}{12}n + \Omega(n^{2/3})$. The proof of the lower bound requires a complex graph theoretical lemma of independent interest: an order-n graph all of whose vertices have degree at most 3 may be separated into components of size at most r vertices by deleting at most $(1/4 + O(1/r))n$ vertices. We also prove that the fraction $1/4$ is optimal.

We also consider how many faults may be tolerated before it is impossible to complete diagnosis in 3 rounds. We show that 3-round diagnosis is impossible if $t \geq \frac{8}{3}\sqrt{n}$ and is possible if $t \leq \sqrt{n/2}$. We also prove that if $t < 0.03n$ then diagnosis can be completed in 4 rounds for sufficiently large n.

1.1.2 Dynamic diagnosis

One way to make the model more realistic is to allow for processors to be repaired. We suppose that a processor is repaired by replacing it with a good processor.

The concept of diagnosis with repair has its roots in *sequential diagnosis,* as proposed by Preparata, Metze and Chien [46]. They said that a testing graph would diagnose a system using sequential or k*-step* diagnosis if performing the tests indicated by the graph would allow at least one faulty processor to be identified. The entire system would be diagnosed by repairing the known faulty processor, and then repeating the testing until there no more faulty processors were detected. This model was studied further in [42,35].

Blough and Pelc show [19] that under a probabilistic testing model, on many networks $O(n \log n)$ tests suffice to complete diagnosis. Their algorithm works by identifying a "fault-free core", i.e., a set of adjacent processors that are almost certainly fault free. Diagnosis proceeds in several rounds, working out from the core. Faulty processors are replaced until the entire system is rendered fault free.

The models for fault diagnosis that we have discussed so far are *static* models; the status of each processor is fixed during the diagnosis. We say that a model is *dynamic* if we permit processors to change status during the diagnosis. A processor could fail, or it could be replaced with a working processor. In general, dynamic diagnosis is more difficult than static diagnosis because we cannot use an algorithm which works by first identifying some good processors and then using them to diagnose the remaining processors.

The *ongoing fault diagnosis* model (defined formally in Section 3.1) investigates a situation in which processors continue to fail indefinitely. We show that it is possible to achieve an *equilibrium* in which we discover and repair faulty processors at the same rate as processors are breaking down. We find an upper bound K on the number of faulty nodes that could ever be simultaneously present in the system. This contrasts with self-stabilizing algorithms, as in [34], where the focus is on showing that the system will return to a diagnosable state if processors stop breaking down, and with the model of [19] where processors do not fail during the diagnosis, so the algorithm halts as soon as all the processors have been repaired.

Because we are interested in the equilibrium state, we define t to be an upper bound on the number of processors that can break down each round, and also to be an upper bound on the number that can be repaired in each round. If we never repair a processor unless we can prove that it is faulty, then in the long run we repair processors at the same rate as they are breaking down. So nothing is gained if we permit a faulty processor to be repaired as soon as it is diagnosed. If in some round we are able to diagnose more than t faulty processors, we set the extra faulty processors to one side, and repair them at the rate of t processors per round.

We will show two major results. First we will show, by presenting an algorithm, that we can take $K = O(t \log t)$, independent of n. That is, that regardless of the number of processors it is possible to ensure that at any time at most $O(t \log t)$ processors are simultaneously faulty.

Second we will show that up to a constant factor this bound is the best possible. An adversary can force at least $t \log_2 t$ simultaneously faulty processors.

1.1.3 Distributed diagnosis

Another major area of work in system level fault diagnosis is *distributed* diagnosis. The first distributed diagnosis model was defined by Kuhl and Reddy [39]. In distributed diagnosis each processor is running its own autonomous program. It is able to view only its neighbors in a fixed interconnection graph. Instead of there being a controller, each processor attempts to build up its own local copy of the system diagnosis using tests that it can perform and information passed on to it by processors that it has previously checked. Examples of distributed diagnosis algorithms can be found in [16,17,34,40,47].

The problem of distributed diagnosis is complicated by the manner in which time is measured. We may either suppose that all processors are running at the same speed, which is called *synchronous* time, or that the processors are not necessarily running at the same speed, which is called *asynchronous* time. In the latter case the time performance of an algorithm may be drastically affected by which processors are fastest. We can view a system running asynchronously as being under the control of an adversary that acts as the scheduler for the system.

Motivated by the cooperative collect problem [49] from distributed computing, we give an algorithm that we prove nearly optimal in the number of tests that it performs, regardless of the choices of the scheduler. With high probability our algorithm can complete diagnosis in $O(n \log^3 n)$ tests. In contrast, the best deterministic algorithm [2] for this problem uses $O(n\sqrt{n})$ tests. It is clear that at least $\Omega(n)$ tests are needed.

The algorithm of Bagchi and Hakimi [9] is an optimal distributed algorithm in terms of the number of transmissions it takes for each processor to have a correct diagnosis of the system. But it doesn't attempt to deal with dynamic failures. In fact since the algorithm works by building rooted spanning trees, passing the diagnostic information to the root and then rebroadcasting it, the algorithm would be especially vulnerable to the root breaking unexpectedly. Also the algorithm would never complete diagnosis if it faced a malicious asynchronous scheduler.

There is already some work that deals with dynamic failure [34,17,16]. However, those papers only show that if after some time there are no further failures then the diagnosis held by each node will converge to the true diagnosis. This is a weak form of *self-stabilization* [23], and it depends, like all self-stabilizing algorithms, on the assumption that malfunctions are separated by sufficiently long error-free periods.

Because the time to complete diagnosis is dependent on the scheduler it is hard to measure the efficiency of an asynchronous distributed algorithm. We use online analysis techniques developed by Ajtai, Aspnes, Dwork, and Waarts [2,7] to demonstrate that our algorithm is close to optimal in time also. That is, we show that a *champion* algorithm written especially for a given schedule does only a log-factor better than our algorithm on the same schedule.

We also analyze how our algorithm would perform under a synchronous model. We show that it completes diagnosis in $O(\log^2 n)$ rounds of testing.

1.2 Other Models

Because we are interested in studying the fundamental problems of communication that underlie the fault diagnosis problem, we have deliberately chosen to concentrate on the simpler variants of the PMC model. In particular we make the following two assumptions:

- Every processor may test any other processor. That is, we are assuming that the network in which the processors are placed is something like an ethernet. However, much of the work in fault diagnosis, going back to [46], assumes that the tests are taking place on a network in which processors may only test their neighbors.

- Good processors never make mistakes. That is, we assume that the diagnosis of a good processor is always accurate. However there is a considerable literature on *probabilistic* diagnosis in which processors are supposed to make mistakes in some independent fashion.

  For example in [19] Blough and Pelc suppose that good processors always accurately test good processors, but are only accurate with probability q when testing *intermittent* faulty processors. They also have *permanent* faulty processors which fail every test performed on them by a good processor.

In general, we give the adversary as much power as possible. The adversary decides where the faulty processors are initially placed, and decides their test results. In dynamic algorithms, the adversary decides which processors fail in each round. In asynchronous algorithms, the adversary decides the schedule. This can be contrasted for example with [19], in which the faulty processors are initially placed at random.

1.3 Definitions

There are many different models for fault diagnosis used in this document. Here we list the properties that are common to all of the models.

- The computing system is considered to be partitioned into n complex indivisible units called *processors* or *nodes*.

- At any time each node has a *status*: either *good* or *faulty*. A good node does not make any errors; it can be relied on to follow its algorithm precisely. We make no assumptions about the behavior of faulty processor.

  However, if a node refuses to follow a protocol (e.g. to perform a test) then it must be faulty. Since having a node reveal its status in this manner can only make the diagnosis easier, we will assume that faulty nodes never do this. If an algorithm asks a faulty node to perform a test, then it will do so. But it may lie about the result of the test.

- A *test* is an interaction between two processors, a *tester* and a *testee*. The tester claims to determine the status of the testee. If the tester is good then it will correctly diagnose the testee. But if the tester is faulty then it has the option of incorrectly diagnosing the testee. A processor is not permitted to test itself. These possible results of a test are summarized in Figure 1.1.

  We say that a processor *participates* in a test if it is either the tester or the testee of the test. A processor may not participate in two tests simultaneously. A test can be viewed as a directed arc pointing from the tester to the testee.

- We call a test *good* if and only if the tester reports that the testee was good. Alternatively we will say that the tester *liked* the testee.

The objective of fault diagnosis is to determine the status of each processor, as accurately as possible, by performing as few tests as possible between the processors.

Outcome of test	testee good	testee faulty
tester good	**good**	**faulty**
tester faulty	**either**	**either**

Figure 1.1: Chart showing possible outcomes of a test

In the following chapters we will define several models of fault diagnosis. These models will specify when tests may be performed, and how the tester and testee for each test are selected. There will usually be a restriction on the number of faulty processors in the system. Some of the models permit processors to change status during the diagnosis.

We will briefly summarize the notation that we will be using to represent graphs. For other graph notation not covered here, see any standard text on the subject, e.g. [15].

- A graph $G = (V, E)$, is a collection of vertices V, and of edges E between them. We only consider simple graphs; there are no loops or parallel edges in any of our graphs.

- Usually a graph will be *undirected.* That is, each edge in the graph can be viewed as a line connecting two vertices. We will also consider *directed* graphs in which each edge can be viewed as an arrow pointing from one vertex to another vertex.

  If G is a directed graph, then $\overline{G}$ represents the equivalent undirected graph. That is, there is an edge between u and v in $\overline{G}$ if and only if there is an edge from u to v, or an edge from v to u in G. Even if there are two edges in G between u and v, one pointing one way, and one pointing the other way, $\overline{G}$ only contains a single edge between u and v.

- The *order* of a graph is the number of vertices in the graph. It will be denoted as $|V|$ or $|G|$.

- The *degree* of a vertex is the number of edges connected to the vertex. We shall say that a graph is r-*regular* if every vertex in the graph has degree exactly r. We shall say that a graph is r-*semiregular* if every vertex in the graph has degree at most r.

- A *path* is a sequence of distinct vertices such that there is an edge in the graph between each vertex and its successor in the sequence. A *cycle* is a path containing at least 3 vertices with the additional property that the first and last vertex in the path are adjacent. A *directed path* is a path in a directed graph, in which each of the edges points to its successor vertex.

- By *subgraph* we always mean an induced subgraph. A subgraph of G is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' = E \cap (V' \times V')$. A *(connected) component* of a graph is a maximal subgraph such that there is a path between every pair of vertices in the subgraph. A *strongly connected component* of a graph is a maximal subgraph such that there is a directed path between every ordered pair of vertices in the subgraph.

Chapter 2

Static Fault Diagnosis

In this chapter we consider the static model of fault diagnosis. Each processor has a fixed status, and we wish to see how many rounds of testing are needed to carry out diagnosis. The chapter consists of several algorithms to solve the problem (which provide upper bounds) and several strategies for the adversary which defeat any algorithm (and thus provide lower bounds.) The algorithms and strategies differ in the assumptions they make about the relationship between n, the number of processors, and t, the upper bound on the number of faulty processors.

The two main results are an algorithm to complete diagnosis in 10 rounds against any adversary, and a proof that if $t > .42n$, and n is sufficiently large then at least 5 rounds are needed for diagnosis. We illustrate the techniques that are used for these results by proving simpler results that are valid for smaller t. These results, which among other things show how many faults may be tolerated in 3-round diagnosis, are of independent interest.

The lower-bound proof requires a result which roughly states that a 3-semiregular graph can be separated into components of a bounded order by deleting around 1/4 of its vertices. This result is proved in Section 2.5. We also prove that the fraction 1/4 is the best possible.

2.1 Definitions

We formalize the *parallel* or *static* model of fault diagnosis below:

- We view the computing system as being made up of n computing units that we will refer to as *processors* or *nodes*.

- The object of the algorithm is to perform complete diagnosis, that is to determine the *status* of all the processors. A processor may be either *good* or *faulty*. A processor's status doesn't change as the algorithm is being performed.

- t is a given upper bound on the number of faulty nodes.

- A *test* is an interaction between two processors, a *tester* and a *testee*. The tester claims to determine the status of the testee. If the tester is good then it will correctly diagnose the testee. But if the tester is itself faulty then it has the option of incorrectly diagnosing the testee. A processor is not permitted to test itself. If the tester reports that the testee was good we shall say that the tester likes the testee, or that the *test was good*.

- In a *round* of testing, several tests may be performed simultaneously, but each processor can participate in at most one test. (A round of tests is a directed matching on the processors, but not necessarily a perfect matching.)

- It is permitted for the tests performed in a particular round to be chosen in the light of the results of tests from former rounds. A round is called *adaptive* if the tests performed in the round are determined using information that was not used to determine the tests performed in any of the previous rounds.

- Let T_i denote the directed graph which has an edge from node a to node b iff processor a tested processor b at some point during the first i rounds of testing. Let $\overline{T}_i$ denote the undirected graph which has an edge between node a and node b iff processor a tested or was tested by processor b at some point during the first i rounds.

Let G_i and $\overline{G}_i$ denote the subgraphs obtained from T_i and $\overline{T}_i$ by restricting the edge set to consist solely of those edges that correspond to good tests.

At times we will find it convenient to view the testing process as a contest between a deterministic *controller* and an omniscient *adversary*. The controller decides which tests to perform, and the adversary selects the faulty nodes in such a way as to maximize the number of rounds needed to complete the diagnosis.

It's easy to see [46] that $t < n/2$ is a necessary condition for diagnosis to be possible at all. We shall refer to the case $t = \lceil \frac{n}{2} \rceil - 1$ as the *general* problem.

2.2 Supernodes

Recall from the last section that G_i denotes the directed graph of all good tests performed in the first i rounds of testing. Then a subset U of the processors, is defined to be a *supernode* in round $i + 1$, if G_i induces a strongly connected graph on U.

Lemma 1 *If U is a supernode in any round then all the processors in U have the same status.*

Proof: Suppose that not all the nodes in U are good. Then there exists $u_0 \in U$ such that u_0 is faulty. Let $u \in U$. Since the graph of good tests on U is strongly connected we can find a sequence of nodes $u = u_r, u_{r-1}, \ldots, u_0$ such that u_j likes u_{j-1} for $1 \leq j \leq r$.

Then u_1 must be faulty, since it likes u_0 which is known to be faulty. Similarly u_2 likes u_1 and so it is faulty. So by induction u_r is also faulty.

Hence if U contains any faulty nodes, then every node in U must be faulty. $\square$

We shall refer to a supernode as *good* if its elements are all good. We shall refer to a supernode as *faulty* if its elements are all faulty.

In [13] an algorithm is presented which performs fault diagnosis. This algorithm works in two phases. First we will give a brief overview of the algorithm. Note that in [13] a supernode is referred to as a mutual admiration society (MAS).

In the first phase all the tests indicated by a fixed testing graph are performed. It is possible to choose the graph so that no matter how the faulty nodes are arranged it will induce a strongly connected component on a large fraction of the good nodes.

Since every good node likes every other good node this component will be a supernode. There might also be other large supernodes containing faulty nodes and at this point there is no way to recognize the status of each supernode.

A second phase of tests is carried out in which several tasks are performed simultaneously:

- Each large supernode tests every other large supernode.
- Every large supernode tests all the processors which are not in a large supernode.
- Every processor not in a large supernode tests each large supernode.

At this point the graph of all tests performed must induce a strongly connected subgraph on all the good nodes. So there will be a supernode containing all the good nodes. Since a strict majority of the nodes are good this supernode can be recognized. So diagnosis is complete.

2.2.1 Extending the algorithm

In this subsection we will describe the idea which lets us improve on the algorithm from [13]. We make two observations.

First we notice that a supernode U can be viewed as a single processor (because all the processors in it have the same status) that can participate in $|U|$ tests in a single round (because all the processors in it can test or be tested independently). This is the reason for the name “supernode”.

Second we observe that the two phases of the [13] algorithm outlined in the last section are nonadaptive. The schedule of tests to be carried out in the first phase is determined before any of these tests are performed. Then the controller can examine the results of the first phase tests and use them to determine the schedule of tests carried out in the second phase. At the end of the second phase the controller can announce the diagnosis of each processor.

These two observations suggest the following algorithm:

1. Carry out several rounds of testing with the objective to construct lots of small supernodes. As many nodes as possible should be contained in some supernode.

2. Apply the [13] algorithm to these supernodes, treating each supernode as if it were a node in the algorithm. By the first observation several rounds of nonadaptive tests on ordinary nodes can be performed in a single round of testing when they are applied to supernodes. So by the second observation this stage of the algorithm can be carried out in 2 rounds.

3. Clean up. Use the supernodes whose status has been determined in stage 2 to diagnose those nodes that were not put into supernodes in stage 1.

This algorithm improves on [13] by taking only 2 rounds to apply stage 2. However it has the additional costs of stages 1 and 3. The number of rounds needed to carry out general diagnosis using this approach is less than a third of the number needed under the former approach.

2.2.2 Diagnosis with at most 3% faulty processors

In this section we shall demonstrate the idea from Section 2.2.1 by proving that if $t \leq 0.03n$ then four rounds are sufficient to complete the diagnosis, for sufficiently large n. See Algorithm 2.1 for the notation and details.

Input: n nodes, at most ϵn of which are faulty, and an integer d.

Output: The status of each node.

Method:

Rounds 1–2 Divide the nodes into cycles of size $2d$, setting aside at most $2d-1$ nodes. Have each processor test the status of its successor in its cycle. Since there is an even number of processors in each cycle this can be done in two rounds.

Any cycle in which every test was good can be viewed as a supernode. Let m be the the number of supernodes.

Round 3 Each supernode can participate in $2d$ testing operations during this round. So we can apply a d-regular testing graph H_d to the supernodes.

H_d will induce a strongly connected component on some subset of the good supernodes.

Round 4 This good component now tests every processor whose status is still in doubt. (We assume that it is large enough for this to take only one round.)

Algorithm 2.1: Four-round diagnosis

We have to show how to choose d and H_d so that this procedure will always work, i.e., so that the strongly connected good component contains more than half of the original nodes.

We will use the same construction for H_d as is used in [13]. Let V be a set of m nodes. H_d is formed by taking the union of d randomly chosen directed Hamiltonian cycles on V. We will show that with high probability an H_d constructed in this manner has the required properties. Hence we can be sure that a suitable test graph exists.

Note that n is original number of processors in the graph and m is the number of supernodes. The value of m is not known until after round 2 has been completed.

We will use the following result to prove the existence of large strongly connected components. It is a natural extension of a lemma of Erdős and Rényi, (see for example [45, p. 45]).

Lemma 2 *Let $G = (V, E)$ be a directed graph of order m. Let $0 < \lambda, \gamma < 1$. Suppose* *that for every pair of sets $A, B \subseteq V$ satisfying $A \cap B = \emptyset$, $|A| + |B| = \lambda m$ and $|A|, |B| \leq \frac{1+\gamma}{2}\lambda m$, there are edges in G directed from A to B and from B to A. Then G induces a strongly connected component of size at least $\gamma\lambda m$ on any subgraph of size at least λm.*

Proof: Let H be any subgraph of G with at least $r = \lambda m$ vertices. Let $C_1, C_2, \ldots C_l$ be the strongly connected components of H.

We may order these components so that if there is an edge $C_i \longrightarrow C_j$ then $i \leq j$. Let

$$k = \max\left\{ j : \sum_{i=1}^{j} |C_i| \leq \frac{1+\gamma}{2} r \right\}.$$

Suppose that $\forall i \, |C_i| \leq \gamma r$. Then we must have

$$\frac{1-\gamma}{2} r \leq \sum_{i=1}^{k} |C_i| \leq \frac{1+\gamma}{2} r.$$

Hence taking A to be $\cup_{i=1}^{k} C_i$ and $B = \cup_{i=k+1}^{l} C_i$, contradicts the premise. So we must have a strongly connected component of the required size. □

In our application of the lemma we interpret λ as the fraction of the m supernodes that are good, and γ as the fraction of the good supernodes that we want to be in a strongly connected component.

Lemma 3 *Let V be a set of m vertices. Let H_d be the union of d independently chosen directed Hamiltonian cycles on V, where any duplicate edges are removed. Let A and B be disjoint subsets of V of size αm and βm respectively. Then*

$$Pr[\nexists \text{ edge } A \rightarrow B \text{ or } \nexists \text{ edge } B \rightarrow A] \leq \left(2 \frac{(m-\alpha m)!(m-\beta m)!}{m!(m-\alpha m-\beta m)!} \right)^d \tag{2.1}$$

Proof: Corollary of [13] Lemma 2. □

Now we will estimate the probability that a randomly chosen H_d satisfies the premise of Lemma 2 for particular values of m, λ and γ. There are $\binom{m}{\lambda m} < 2^m$ ways that the λm good vertices can be chosen from the m vertices. There are at most $2^{\lambda m}$ ways that the good vertices could be partitioned into A and B. Let $\alpha = |A|/m$ and $\beta = |B|/m$. We need only consider partitions which satisfy

$$\frac{(1-\gamma)\lambda}{2} \leq \alpha, \beta \leq \frac{(1+\gamma)\lambda}{2} \quad \text{where} \quad \alpha + \beta = \lambda. \tag{2.2}$$

Examining the right hand side of (2.1) we see it is maximized under the constraints of (2.2) if we take

$$\alpha = \frac{(1+\gamma)\lambda}{2} \quad \text{and} \quad \beta = \frac{(1-\gamma)\lambda}{2}.$$

So by Lemma 3 we have

$$\Pr\left[H_d \text{ satisfies premise}\right] \leq 2^{(1+\lambda)m}\left(2\frac{(m-\alpha m)!(m-\beta m)!}{m!(m-\alpha m-\beta m)!}\right)^d \tag{2.3}$$

Applying Stirling's formula and taking the log of the right hand side gives

$$\begin{aligned} m\Big((1+\lambda)\ln 2 + d(& \quad (1-\beta)\ln(1-\beta) \\ & + \ (1-\alpha)\ln(1-\alpha) \\ & - \ (1-\lambda)\ln(1-\lambda))\Big) + O(1) \end{aligned} \tag{2.4}$$

Now consider Algorithm 2.1, in which n is the original number of nodes, and at most ϵn nodes are faulty. Let μ denote the fraction of nodes that are in a supernode at the start of round 3. We have

$$\mu \geq 1 - 2d\epsilon$$

since each cycle that doesn't form a supernode must contain at least one faulty node. At most an ϵ fraction of the supernodes can be faulty. Let m be the number of supernodes, and let λ be the fraction of the supernodes that are good. Then

$$\lambda \geq \frac{\mu - \epsilon}{\mu} = 1 - \frac{\epsilon}{\mu} \geq 1 - \frac{\epsilon}{1-2d\epsilon} \tag{2.5}$$

To complete the diagnosis in round 4 we need to find a big strongly connected component in round 3 that contains at least one half of the original n nodes. It is not enough for the component to contain one half of the m supernodes, since not every node will appear in a supernode. So we need $\gamma\lambda\mu n > (1/2)n$ where γ is the fraction of good supernodes that are in the big strongly connected component. Equivalently we need

$$\gamma > \frac{1}{2(1-(2d+1)\epsilon)} \tag{2.6}$$

If the right hand side of (2.3) is smaller than 1, where λ and γ given by (2.5) and (2.6), then a suitable H_d will exist by Lemmas 2 and 3. Alternatively we need the coefficient of m in (2.4) to be negative, in which case a suitable graph will exist for large enough m (and hence for large enough n).

Numerical calculations show that if we take $d = 5$ and $\epsilon = .03$ then we get $\lambda > .96$, $\gamma > .75$ and the coefficient of m in (2.4) is at most -0.021. So we have

Theorem 4 *Algorithm 2.1 can be used to complete fault diagnosis in 4 rounds provided that at most 3% of n processors are faulty and n is sufficiently large.*

2.3 General Problem

In this section we are interested in the general fault diagnosis section. The only restriction on the number of faulty processors t is that $t < n/2$. We give an algorithm for complete diagnosis which substantially lowers the constant number of testing rounds needed from 32 given in [13], to 10. This algorithm is an expansion of the algorithm that we originally presented in [11].

As in Section 2.2.1 the algorithm works by building supernodes. We will proceed as follows: First we build up supernodes, using a pairing operation. Then we apply the existing diagnosis algorithm to the supernodes and diagnose them. Then we have some nodes whose status has been determined to be good. We use these nodes to diagnose all the nodes that were set aside as we were building the supernodes. Since it is possible that almost all of the nodes were set aside, we will need to exploit information gained about the arrangement of faulty nodes during the "construction of supernode" phase in order to complete the diagnosis quickly.

Input: n processors, a strict majority of which as good.

Output: The status of every processor

Method:

Round 1	test a perfect matching	
Round 2	build supernodes of size 2	build chains
Round 3	build supernodes of size 4	build chains
Round 4	build supernodes of size 8	build chains
Round 5	build supernodes of size 16	build chains
Round 6	build supernodes of size 32	build chains
Round 7	apply test graph to supernodes	build chains
Round 8	diagnose supernodes	build chains
Round 9	tests chains with good supernodes	
Round 10	test all remaining undiagnosed nodes	

Algorithm 2.2: General diagnosis summary

Algorithm 2.2 is a summary of this algorithm. It shows which tasks are performed in which rounds. The actual algorithm as described in this section is made more complex because we have to cover several exceptional cases, eg. when there are an odd number of nodes. Our aim here is to find an algorithm which works for all values of n, not just for n sufficiently large.

2.3.1 General diagnosis algorithm

In this section we will describe in detail the tests that are performed in each round of the algorithm. We will defer the proof of correctness to the following section.

First we shall make some definitions that will be used in the description of the algorithm. Let n be the number of processors. During the algorithm we will find it convenient to partition the processors into four classes: $\mathcal{S}$, $\mathcal{J}$, $\mathcal{C}_1$ and $\mathcal{C}_2$.

$\mathcal{S}$: processors that are placed in order-32 supernodes and are eventually diagnosed using the algorithm from [13].

$\mathcal{J}$: processors that are used as tie-breakers in cases when exactly 50% of the processors in $\mathcal{S}$ are good. They are referred to as judges.

$\mathcal{C}_1$: processors that were discarded in pairs and are then arranged into a chain. To be arranged in a chain means that tests are carried out in both directions between the pairs using the pattern shown in figure 2.3. Each row of the figure corresponds to one pair.

$\mathcal{C}_2$: member processors of supernodes (of order larger than 1) that were discarded in pairs. These supernodes are also arranged into a chain. This chain is disjoint from the $\mathcal{C}_1$-chain.

Round 1 Pair the nodes off. If n is odd place the single unpaired node in $\mathcal{J}$. For each pair (a, b) of nodes let processor a test processor b; if a does not like b, place (a, b) in $\mathcal{C}_1$. Let $\{a_1, b_1\}, \ldots, \{a_{r_1}, b_{r_1}\}$ be the pairs of supernodes that are placed in $\mathcal{C}_1$.

Round 2 For each pair (a, b) from round 1 that is not in $\mathcal{C}_1$, let b test a. If b likes a, then we can consider the pair to be a single order-2 supernode. Otherwise we place the pair in $\mathcal{C}_1$.

Let $\{c_1, d_1\}, \ldots, \{c_{r_2}, d_{r_2}\}$ be the pairs of supernodes that are placed in $\mathcal{C}_1$. Note that each processor c_i is now known to be faulty.

For $1 \leq i < r_1$, i odd, let processor a_i test processor a_{i+1}.

For $1 \leq i < r_1$, i odd, let processor b_i test processor b_{i+1}.

Round 3 Pair off the order-2 supernodes from round 2. If there is an odd number of them, place the unpaired supernode in $\mathcal{J}$. Since the supernodes can engage in two testing operations per round, only one round is needed to test both ways within each pair. If each member of a pair likes its mate, then we can consider the pair to be a

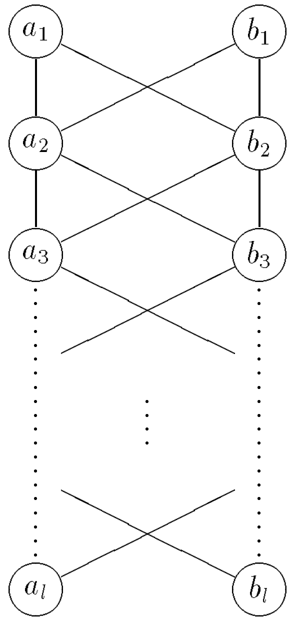


Figure 2.3: Chain testing pattern

single order-4 supernode. Otherwise we place the pair of order-2 supernodes in $\mathcal{C}_2$. The sequence of tests needed to form a supernode of order-4 is shown in Figure 2.4. Of course the supernode is only obtained if every test shown is good.

Let $\{C_1^2, D_1^2\}, \ldots, \{C_{r_3}^2, D_{r_3}^2\}$ be the pairs of supernodes that are placed in $\mathcal{C}_2$.

For $1 \leq i < r_1$, i odd, let processor a_i test processor b_{i+1}.

For $1 \leq i < r_1$, i odd, let processor b_i test processor a_{i+1}.

For $1 \leq j < r_2$, $j \equiv r_2 - 1 \pmod 2$, let processor d_j test processor d_{j+1}.

Round 4 Pair off the order-4 supernodes from round 3. If there is an odd number of them, place the unpaired supernode in $\mathcal{J}$. Since the supernodes can engage in four testing operations per round, only one round is needed to test both ways within each pair. If each member of a pair likes its mate, then we can consider the pair to be a single order-8 supernode. Otherwise we place the pair of order-4 supernodes in $\mathcal{C}_2$.

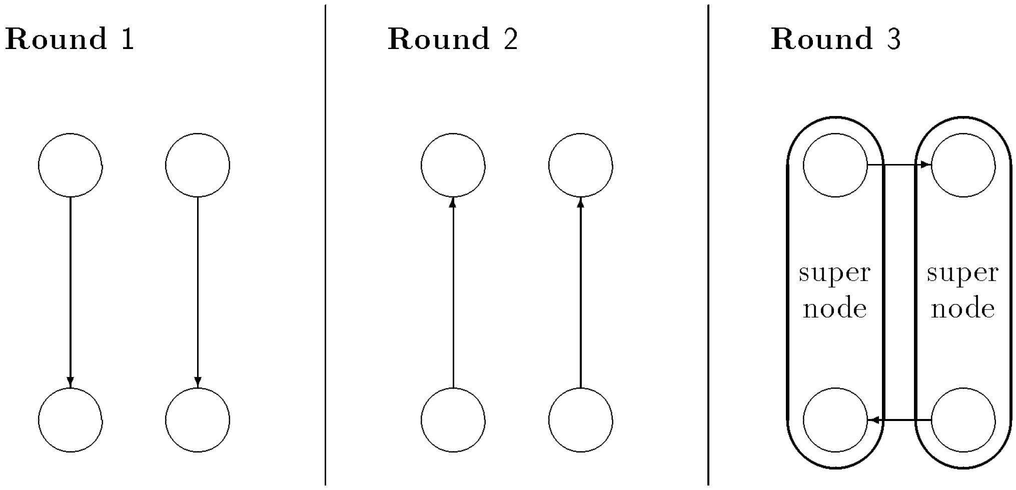


Figure 2.4: Building supernodes of size 4

Let $\{C_1^4, D_1^4\}, \ldots, \{C_{r_4}^4, D_{r_4}^4\}$ be the pairs of supernodes that are placed in $\mathcal{C}_2$.

For $1 < i < r_1$, i even, let processor a_i test processor a_{i+1}.

For $1 < i < r_1$, i even, let processor b_i test processor b_{i+1}.

For $1 \leq j < r_2$, $j \equiv r_2 \pmod 2$, let processor d_j test processor d_{j+1}.

If $r_1 > 0$ and $r_2 > 0$ let processor d_{r_2} test processor b_1.

For $1 \leq i < r_3$, i odd, let supernode C_i^2 test supernodes C_{i+1}^2 and D_{i+1}^2.

For $1 \leq i < r_3$, i odd, let supernode D_i^2 test supernodes C_{i+1}^2 and D_{i+1}^2.

Round 5 Pair off the order-8 supernodes from round 4. As before place the unpaired supernode (if any) in $\mathcal{J}$. Each pair either forms an order-16 supernode or is placed in $\mathcal{C}_2$. Let $\{C_1^8, D_1^8\}, \ldots, \{C_{r_5}^8, D_{r_5}^8\}$ be the pairs that are placed in $\mathcal{C}_2$.

For $1 < i < r_1$, i even, let processor a_i test processor b_{i+1}.

For $1 < i < r_1$, i even, let processor b_i test processor a_{i+1}.

If $r_1 > 0$ and $r_2 > 0$ let processor d_{r_2} test processor a_1.

For $1 < i \leq r_3$, i even, let supernode C_i^2 test supernodes C_{i-1}^2 and D_{i-1}^2.

For $1 < i \leq r_3$, i even, let supernode D_i^2 test supernodes C_{i-1}^2 and D_{i-1}^2.

For $1 \leq i < r_4$, let supernode C_i^4 test supernodes C_{i+1}^4 and D_{i+1}^4.

For $1 \leq i < r_4$, let supernode D_i^4 test supernodes C_{i+1}^4 and D_{i+1}^4.

Round 6 Pair off the order-16 supernodes from round 5. As before place the unpaired supernode (if any) in $\mathcal{J}$. Each pair either forms an order-32 supernode, in which case it is placed in $\mathcal{S}$, or it is placed in $\mathcal{C}_2$. Let $\{C_1^{16}, D_1^{16}\}, \ldots, \{C_{r_6}^{16}, D_{r_6}^{16}\}$ be the pairs that are placed in $\mathcal{C}_2$.

The partition of processors into $\mathcal{S}$, $\mathcal{J}$, $\mathcal{C}_1$ and $\mathcal{C}_2$ is now complete.

The set $\mathcal{J}$ may contain supernodes with orders 1, 2, 4, 8 and 16. It may have at most one supernode of each order. If $\mathcal{J} \neq \emptyset$, let J be the largest supernode in $\mathcal{J}$ and let J simultaneously test all the smaller supernodes in $\mathcal{J}$.

We now wish to reverse all the tests that were made between a_i, b_i and d_j nodes in $\mathcal{C}_1$. The objective of these tests is to form larger supernodes, so we only need to reverse the good tests. Also if a node tested both of the nodes in the pair that follows it, and liked both of them, then the tester is known to be faulty (since each pair must contain a faulty node) and we do need to carry out any more tests involving a faulty node. Thus each node has at most one reverse test we wish to perform on it.

These restrictions imply that the graph of tests we wish to perform among the $\mathcal{C}_1$ nodes is a forest in which each vertex has degree at most 3. So it is the union of three directed matchings. Thus we may perform the reverse tests in three rounds. Carry out the first round of reverse tests now.

For $1 < i \leq r_4$, let supernode C_i^4 test supernodes C_{i-1}^4 and D_{i-1}^4.

For $1 < i \leq r_4$, let supernode D_i^4 test supernodes C_{i-1}^4 and D_{i-1}^4.

For $1 \leq i < r_5$, let supernode C_i^8 test supernodes C_{i+1}^8 and D_{i+1}^8.

For $1 \leq i < r_5$, let supernode D_i^8 test supernodes C_{i+1}^8 and D_{i+1}^8.

For $1 < i \leq r_5$, let supernode C_i^8 test supernodes C_{i-1}^8 and D_{i-1}^8.

For $1 < i \leq r_5$, let supernode D_i^8 test supernodes C_{i-1}^8 and D_{i-1}^8.

Round 7 Let N be the number of order-32 supernodes in $\mathcal{S}$. We perform tests between the supernodes in $\mathcal{S}$. These tests will result in the formation of giant supernodes. We will call a supernode *giant* if it contains strictly more than $N/6$ of the N original order-32 supernodes. We consider three cases:

- If $N = 0$ then $\mathcal{S} = \emptyset$ and there are no giant supernodes. In this case $\mathcal{J} \neq \emptyset$ and J is good. Any other supernodes in $\mathcal{J}$ were diagnosed by J in round 6.
- If $1 \leq N \leq 16$, then every order-32 supernode tests every other order-32 supernode. This requires at most 30 testing operations per supernode. Because at least $N/2$ of the order-32 supernodes are good, and thus will like each other, there will be at most 3 giant supernodes.

  If $\mathcal{J} \neq \emptyset$ let J test one of the supernodes in $\mathcal{S}$.
- If $N > 16$ then apply a test graph H_{16} to $\mathcal{S}$. This graph will require each supernode to perform 16 tests, and to be tested 16 times by other supernodes. The construction of H_{16} will be given in Section 2.3.2.

Let $A_1, A_2, \ldots, A_l$ be the giant supernodes, in order of descending size. Since the supernodes are disjoint, $l \leq 5$. Let $B = \mathcal{S} \setminus (A_1 \cup A_2 \cup \ldots \cup A_l)$.

Perform the second round of reverse tests among the a_i, b_i and d_i nodes in $\mathcal{C}_1$.

For $1 < i < r_3$, i even, let supernode C_i^2 test supernodes C_{i+1}^2 and D_{i+1}^2.
For $1 < i < r_3$, i even, let supernode D_i^2 test supernodes C_{i+1}^2 and D_{i+1}^2.
For $1 \leq i < r_6$, let supernode C_i^{16} test supernodes C_{i+1}^{16} and D_{i+1}^{16}.
For $1 \leq i < r_6$, let supernode D_i^{16} test supernodes C_{i+1}^{16} and D_{i+1}^{16}.
For $1 < i \leq r_6$, let supernode C_i^{16} test supernodes C_{i-1}^{16} and D_{i-1}^{16}.
For $1 < i \leq r_6$, let supernode D_i^{16} test supernodes C_{i-1}^{16} and D_{i-1}^{16}.

Define the C^{16} chain to be the chain made up of order-16 supernodes (both the C_i^{16} supernodes and the D_i^{16} supernodes), the C^8 chain to be the chain of order-8 supernodes and so on. Define two partial functions δ and δ^{-1}. Let $\delta(2)$ be the order of the lowest-order chain, whose order is larger than 2, and which is nonempty. Thus $\delta(2)$ is 4 if the C^4 chain is nonempty. Otherwise $\delta(2)$ is 8 if the C^8 chain is nonempty. Otherwise $\delta(2) = 16$ if the C^{16} chain is nonempty. $\delta(2)$ is undefined if the C^4, C^8 and C^{16} chains are all empty. Define $\delta(4)$ and $\delta(8)$ in a similar manner. Also define $\delta^{-1}(16)$, $\delta^{-1}(8)$, and $\delta^{-1}(4)$ to be in each case the largest-order nonempty chain which has a smaller order than the argument.

If $r_6 > 0$ and $\delta^{-1}(16)$ is defined, let supernodes $C^{16}_{r_6}$ and $D^{16}_{r_6}$ test $C_1^{\delta^{-1}(16)}$ and $D_1^{\delta^{-1}(16)}$.

If $r_5 > 0$ and $\delta^{-1}(8)$ is defined, let supernodes $C^8_{r_5}$ and $D^8_{r_5}$ test $C_1^{\delta^{-1}(8)}$ and $D_1^{\delta^{-1}(8)}$.

If $r_4 > 0$ and $\delta^{-1}(4)$ is defined, let supernodes $C^4_{r_4}$ and $D^4_{r_4}$ test $C_1^{\delta^{-1}(4)}$ and $D_1^{\delta^{-1}(4)}$.

Round 8 We now finished diagnosing $\mathcal{S}$. We have the same cases as in round 7:

- If $N = 0$ then $\mathcal{S} = \emptyset$.

- If $1 \leq N \leq 16$, then there is exactly one good giant supernode. If $|A_1| > |A_2|$ then it is A_1. If $|A_1| = |A_2|$ then $|A_1| = N/2$ and $\mathcal{J} \neq \emptyset$. If J liked the order-32 supernode it tested in round 7, then the giant supernode that it belongs to is good. If J did not like the order-32 supernode that it tested in round 7, then the giant supernode that it belongs to is faulty. In each case we identify a good giant supernode, A_ρ say. All the processors in $\mathcal{S} \setminus A_\rho$ are faulty.

  Let A_ρ test all the supernodes in $\mathcal{J}$. Diagnosis of $\mathcal{S} \cup \mathcal{J}$ is complete.

- If $N > 16$, then let each giant supernode test and be tested by every other giant supernode. This requires the use of at most 8 processors from each giant supernode.

  Also let each supernode in B, the smaller supernodes from $\mathcal{S}$, test and be tested by each giant supernode. Since $|B| < 5|A_i|$ these tests require the use of at most 10 processors from each order-32 supernodes that was assigned to $\mathcal{S}$.

If $\mathcal{J}$ contains an order-16 supernode let it be tested by each A_i. Do the same for an order-8 supernode, if there is one present in $\mathcal{J}$.

If $l = 1$, i.e. there is only one giant supernode, then it must be good. Let it test all the supernodes in $\mathcal{J}$. Otherwise if $l > 1$ and $\mathcal{J} \neq \emptyset$ let J test A_1 instead.

Even if it was possible for a supernode of order-32 to be a giant supernode, this supernode would need to carry out at most 8 testing operations (to test other giant supernodes), at most 10 operations (to test supernodes in B) and at most 5 operations (to test or be tested by each supernode in $\mathcal{J}$). This gives a total of at most 23 testing operations, so all of these operations may be carried out simultaneously by the order-32 node.

All of $\mathcal{S}$ is now diagnosed. All but 7 processors from $\mathcal{J}$ have been diagnosed, and if there was only one giant supernode then all of $\mathcal{J}$ has been diagnosed.

Perform the third and final round of reverse tests in $\mathcal{C}_1$.

If $\mathcal{C}_1 \neq \emptyset$ let (p, q) be the bottom pair of nodes in the $\mathcal{C}_1$-chain. So if $r_1 > 0$ then $(p, q) = (a_{r_1}, b_{r_1})$, otherwise $(p, q) = (c_{r_2}, d_{r_2})$. If a node $b \in \mathcal{C}_1$ liked both p and q then we would know that b was faulty. So there are at most 2 reverse tests in $\mathcal{C}_1$ in which either p or q participates. Thus, without loss of generality we may assume that p and q are not involved in any reverse test in this round.

If there are two processors that are known to be good, and which have not been allocated a testing operation yet, then use them to diagnose p and q. This will be possible if $N \leq 16$ and there are at least two processors in $\mathcal{S} \cup \mathcal{J}$. It is also possible if $N > 16$ and there is exactly one giant supernode (which must be good).

For $1 < i \leq r_3$, i odd, let supernode C_i^2 test supernodes C_{i-1}^2 and D_{i-1}^2.

For $1 < i \leq r_3$, i odd, let supernode D_i^2 test supernodes C_{i-1}^2 and D_{i-1}^2.

If $r_5 > 0$ and $\delta(8)$ is defined, let supernodes C_1^8 and D_1^8 test the final pair of supernodes in $C^{\delta(8)}$.

If $r_4 > 0$ and $\delta(4)$ is defined, let supernodes C_1^4 and D_1^4 test the final pair of supernodes in $C^{\delta(4)}$.

If $r_3 > 0$ and $\delta(2)$ is defined, let supernodes C_1^2 and D_1^2 test the final pair of supernodes in $C^{\delta(2)}$.

The nodes in $\mathcal{C}_2$ are now arranged in a single chain.

Round 9 If $N > 16$ and $\mathcal{J}$ is not completely diagnosed, test each undiagnosed supernode in $\mathcal{J}$ with a good processor from $\mathcal{S}$.

When the chain pattern was tested among the nodes in $\mathcal{C}_1 \cup \mathcal{C}_2$, it is probable that some pairs of nodes liked each other, and thus formed new supernodes; each new supernode is a union of two or more of the supernodes that were placed in $\mathcal{C}_1 \cup \mathcal{C}_2$. We will refer to these new supernodes as *fragments*. We will also consider a supernode in $\mathcal{C}_1 \cup \mathcal{C}_2$ that didn't like any of its neighbors in the chain to be a fragment.

We will say that a fragment covers a row of a chain if one of the two processors in the row is a member of the fragment. Similarly a set of fragments *covers* a row if one of the fragments in the set covers the row.

Test fragments using all the available processors from $\mathcal{S} \cup \mathcal{J}$ that have been diagnosed as good. Thus the fragments that are tested are also diagnosed. Choose the fragments to test so that they cover as many rows as possible.

More precisely the following greedy algorithm should be used: Start at one end of the $\mathcal{C}_2$-chain and test the fragment in the first row which causes the most rows to be covered. Continue by looking at the first row which has not been covered and diagnosing the fragment from that row which causes the most additional rows to be covered. (Note the fragment which covers the most additional rows might be the *smaller* of the two fragments.) When all of $\mathcal{C}_2$ has been covered start diagnosing fragments of $\mathcal{C}_1$ in the same manner, starting from the top end. (i.e. start from the other end to the end where processors p and q are located.)

Round 10 Use all the processors from $\mathcal{S} \cup \mathcal{J}$ that have been diagnosed as good to test all the remaining fragments in $\mathcal{C}_1 \cup \mathcal{C}_2$ whose status is uncertain. This will complete the diagnosis.

2.3.2 Diagnosing the supernodes

In this section we will prove that the algorithm given in Section 2.3.1 will diagnose the supernodes placed in $\mathcal{S}$ by the end of round 8. We know that a strict majority of the nodes are good. Whenever we place a pair of nodes or supernodes into $\mathcal{C}_1 \cup \mathcal{C}_2$ we know that one member of the pair does not like its mate. Thus at least half the processors removed are faulty. So a strict majority of the processors that remain in $\mathcal{S} \cup \mathcal{J}$ are good.

By the end of round 6 the processors in $\mathcal{S}$ are grouped as supernodes of order-32. $\mathcal{J}$ can contain supernodes of order 1, 2, 4, 8 or 16. It contains at most one node of each order. If $\mathcal{J} \neq \emptyset$, J is defined to be the largest supernode in $\mathcal{J}$.

Either a strict majority of the nodes in $\mathcal{S}$ are good, or exactly half of them are good in which case J has to exist and must be good. In the latter case J serves as a *judge*.

The object of this section is to show how we can diagnose the processors in $\mathcal{S}$ using only two rounds of testing. Let N be the number of order-32 supernodes in $\mathcal{S}$.

We distinguish between two cases. First suppose $N \leq 16$. Then in round 7, every order-32 supernode can test, and be tested by every other order-32 supernode. Simultaneously J tests one of the supernodes.

Since all possible tests between the order-32 supernodes have been carried out, and since the good nodes like each other, they will form a huge supernode containing at least half of the order-32 supernodes. If there is one such huge supernode, it must contain the good processors.

If there are two huge supernodes then we use J to judge between them. If J liked the processor it tested then that processor is in the good huge supernode. Otherwise the processor J tested is in the faulty huge supernode.

In round 8 the diagnosis of $\mathcal{S} \cup \mathcal{J}$ is finished, by testing the supernodes of $\mathcal{J}$ using processors from the good huge supernode. Since the good huge supernode contains at least 32 processors it can simultaneously test p and q. If $N = 0$ then J must be good, and in that case we have already determined the status of the other nodes in $\mathcal{J}$ when J tested them in round 6. Provided that $|J| > 1$, J tests p and q in round 8.

Otherwise we have $N > 16$. In this case we diagnose the supernodes by adapting the algorithm from [13], as described in Section 2.2. This algorithm has two phases. First it performs the tests given by a fixed regular testing graph. The graph chosen is guaranteed to induce at least one *giant* component (a component of size larger than $N/6$) on the good nodes, but it may also induce giant components on the faulty nodes. Second these components test each other, any processors that are not in giant components, and any processors in $\mathcal{J}$.

The test graph, denoted H_{16}, is obtained by randomly selecting 16 Hamiltonian cycles on the same N vertices, and then superimposing them. Each edge is tested both ways, so 32 rounds of nonadaptive testing are necessary and sufficient to perform the tests given by the graph. Combining Lemmas 2 and 3 we have:

Lemma 5 *The probability that a randomly selected graph of type H_d fails to induce at least one large good component of size larger than $N/6$, regardless of how the good nodes are arranged, is at most*

$$P(N) = 2^{3N/2} \left(\frac{(\lfloor \frac{5}{6}N \rfloor)!(\lceil \frac{2}{3}N \rceil)!}{N!(\lceil \frac{1}{2}N \rceil)!} \right)^d$$

So a suitable testing graph, H_{16}, will exist if $P(N) < 1$. [13] proves that this happens for sufficiently large N. We need to show the existence of H_{16} for all $N > 16$, because even if n is large it is possible that most pairs of processors are placed in $\mathcal{C}_1 \cup \mathcal{C}_2$ so that N is small.

We need to show that if $d = 16$ and $N > 16$ then $P(N) < 1$. We shall find an explicit value N_0 such that for N larger than N_0, $P(N)$ is decreasing. Numerical calculation will show that $P(N_0) < 1$.

Because of the floor and ceiling operators, the algebra is affected by the residue modulo 6. For instance if $N = 6k + 1$ we have

$$\left(\frac{P(N)}{P(N-6)}\right)^{\frac{1}{16}} = 2^{9/16}\frac{(5k)_5(4k+1)_4}{(6k+1)_6(3k+1)_3}$$

where $(n)_k = n(n-1)\cdots(n-k+1)$.

If $P(N) < P(N-6)$ then this fraction should be smaller than one. Expanding the polynomials, and using $2^{9/16} < \frac{3}{2}$, we need to show that

$$119424k^9 + 2220864k^8 - 4140336k^7 + \cdots - 3744k^2 > 0.$$

Since the leading positive terms will overwhelm the smaller terms this polynomial is clearly positive for all $k > 20$ say. A similar calculation can be performed for all the residues modulo 6, and it shows that if $N > 6 \cdot 20$ then $P(N)$ is decreasing when compared with $P(N-6)$. Numerical calculation (performed with Mathematica) shows that $P(N) < 1$ for all N between 16 and 120.

The algorithm given in Section 2.3.1 explicitly shows that it is possible in round 8 for the giant supernodes to test each other, and to also test all the processors in $\mathcal{S}$ that are not in a giant supernode. So after round 8 all the good processors in $\mathcal{S}$ must be in the same huge supernode, where we define a supernode to be *huge* if it contains at least half of the processors in $\mathcal{S}$.

As with the $N \leq 16$ case, J is used to test one of the giant components. If exactly half the processors in $\mathcal{S}$ are good, and there are two huge supernodes, then J will indicate which huge supernode is the good one.

We must also diagnose the processors in $\mathcal{J}$. Unfortunately in round 8 there may be up to five giant components, not all of which are good. (Actually tighter analysis shows that there can be at most three giant components.) So a supernode of size 1, 2 or 4 cannot be diagnosed in round 8, since it cannot be simultaneously tested by all the giant supernodes. So up to seven processors from $\mathcal{J}$ may be left until round 9 before they are diagnosed.

When there is only one giant supernode, then it must be good, since there will always be a good giant supernode. In this case the algorithm will diagnose all of $\mathcal{S} \cup \mathcal{J}$ in round 8. It will also diagnose both p and q. In particular if less than one sixth of the processors in $\mathcal{S} \cup \mathcal{J}$ are faulty, then there can be no faulty giant supernodes and so diagnosis will be finished in round 8.

In summary:

Lemma 6 *Let $\mathcal{S}$ and $\mathcal{J}$ be as described above. The algorithm given in Section 2.3.1 will diagnose $\mathcal{S}$ by the end of round 8, but there may be up to 7 processors from $\mathcal{J}$ still undiagnosed at that time. In every case it will completely diagnose $\mathcal{S} \cup \mathcal{J}$ by the end of round 9.*

If $|\mathcal{S} \cup \mathcal{J}| > 1$ and one or more of the following conditions are true:

(a) $N \leq 16$, or

(b) Less than $1/6$ of the processors in $\mathcal{S} \cup \mathcal{J}$ are faulty,

then $\mathcal{S} \cup \mathcal{J} \cup \{p, q\}$ will be diagnosed by the end of round 8.

2.3.3 Building the chains

We wish to arrange the processors in $\mathcal{C}_1 \cup \mathcal{C}_2$ into the chain pattern shown in figure 2.3. For each edge in the pattern we carry out a test both ways. Each row of the pattern should contain a pair of nodes that were discarded together. The chains must be constructed by the end of round 8.

Since each node in a chain is involved with eight tests, it will take up to eight rounds to build the chain. After each of the first six rounds those pairs that fail to like each other need to be added to a chain. The earlier that a pair is discarded, the more rounds there are to carry out the chain tests. However pairs that are discarded later are pairs of supernodes, and so they can be linked together in fewer rounds.

Unfortunately it is not possible to make a single chain from all the pairs in the time available. The problem occurs when some pairs are discarded as simple nodes after round 1, and then there are no more discards until round 6 when some order-16 supernodes are discarded. To make a link between the order-1 and order-16 nodes takes at least 3 rounds. The ability of the order-16 nodes to perform many simultaneous tests doesn't help the order-1 nodes. The algorithm presented in Section 2.3.1 builds two chains: $\mathcal{C}_1$ made from simple nodes and $\mathcal{C}_2$ made from supernodes.

The motivation for using the chain pattern, is to construct larger supernodes, which we call *fragments*. If two adjacent rows both contain a good node, then the good nodes will form a supernode together. Once a tester has shown it doesn't like its testee there is no point in reversing the test. If a node likes both nodes in an adjacent pair, then it is immediately diagnosed as faulty (since one of the nodes it liked was faulty) and so there is no need to test it again, and no point in using it as a tester.

See the algorithm for the details of the construction.

2.3.4 Diagnosing the chains

At the start of round 9 we have some nodes in $\mathcal{S} \cup \mathcal{J}$ that have been diagnosed as good. We wish to use them to diagnose the nodes in $\mathcal{C}_1 \cup \mathcal{C}_2$. We shall improve on an idea from [12].

The nodes in $\mathcal{C}_1$ and those in $\mathcal{C}_2$ have been tested with the chain testing pattern shown in Figure 2.3. The tests divide the chain into a number of new strongly connected components (i.e., larger supernodes), which we shall refer to as *fragments*.

Each row contains at least one faulty node. We will call a row *all–faulty* if both processors in the row are faulty. We will call a row *half–faulty* if only one of the processors in the row is faulty. Whenever two adjacent rows are both half–faulty then the chain pattern will ensure that the two good nodes belong to the same fragment.

Let $m = |\mathcal{S} \cup \mathcal{J}|$, and $k = |\mathcal{C}_1 \cup \mathcal{C}_2|$. Define g to be the number of surplus good nodes in $\mathcal{S} \cup \mathcal{J}$, and b to be the number of surplus faulty nodes in $\mathcal{C}_1 \cup \mathcal{C}_2$, i.e.,

$$\begin{aligned} g &= |\{a \in \mathcal{S} \cup \mathcal{J} : a \text{ good}\}| - |\{a \in \mathcal{S} \cup \mathcal{J} : a \text{ faulty}\}| \\ b &= |\{a \in \mathcal{C}_1 \cup \mathcal{C}_2 : a \text{ faulty}\}| - |\{a \in \mathcal{C}_1 \cup \mathcal{C}_2 : a \text{ good}\}| \end{aligned}$$

Since a strict majority of the nodes are good, $g > b$. Since each all–faulty pair in the chains contributes two surplus faulty nodes, there are exactly $b/2$ all–faulty pairs. So b must be even.

Suppose that $m = 1$. That is, with the exception of a single processor, all the processors are placed in one of the chains. So the single processor must be good. Thus $g = 1$, which implies that $b = 0$, and so there are no all–faulty pairs. So each of the two chains has a good fragment that runs the entire length of the chain. In round 9 the controller uses the known good processor to test a chain fragment from the $\mathcal{C}_2$-chain. Because it uses a greedy algorithm it will test the longest fragment in the chain. So either it will test the good fragment, or if all the faulty processors in the $\mathcal{C}_2$-chain formed a single fragment, then it might test a faulty fragment. But in the latter case the controller can deduce that the other fragment is good. By exactly the same argument the controller will diagnose the $\mathcal{C}_1$-chain in round 10. If either of $\mathcal{C}_1$ or $\mathcal{C}_2$ is empty then the diagnosis will be finished earlier. So we are done in the case that $m = 1$.

Otherwise, consider a row in one of the chains. The row is a pair of supernodes. Since, by construction, the supernodes do not like each other they belong to different fragments. We say that a row is *covered* by a fragment if the fragment includes one of the nodes in the row. In round 9 the controller diagnoses some of the fragments using the known good processors in $\mathcal{S} \cup \mathcal{J}$. We shall prove that by the end of round 9 the controller will have diagnosed fragments that together cover every row.

Now consider the good fragments in the chains. If there are several good fragments in a chain, then there must be an all–faulty pair between them. Since there are at most two chains, there are at most $b/2 + 2$ good fragments. The good fragments will cover all the half–faulty rows. Since there are $b/2$ all–faulty rows, it is possible to cover all the rows by testing $b + 2$ fragments.

An easy induction shows that the greedy algorithm that the controller uses in round 9 to select the fragments to test, will select the minimum number of fragments necessary to cover all the rows. Thus it suffices to show that $b + 2$ fragments have been tested by the end of round 9.

Recall that N is the number of order-32 supernodes in $\mathcal{S}$. We consider two cases:

(a) If $N \leq 16$ or less than $1/6$ of the processors in $\mathcal{S} \cup \mathcal{J}$ are faulty, then by Lemma 6 the controller diagnoses p and q in round 8. Since either the fragment that includes p or the fragment that includes q is in the minimum cover of the chains, the controller only needs to diagnose $b+1$ fragments in round 9 to cover all the chains.

By Lemma 6, all of the good processors in $\mathcal{S} \cup \mathcal{J}$ are diagnosed in round 8. Therefore the controller diagnoses

$$\frac{1}{2}(m+g) \geq g \geq b+1$$

fragments in round 9. So all the rows are covered by the fragments that the controller tests.

(b) If $N \leq 16$ and at least $1/6$ of the processors in $\mathcal{S} \cup \mathcal{J}$ are faulty then, by Lemma 6, all except possibly 7 of the good processors in $\mathcal{S} \cup \mathcal{J}$ will have been diagnosed by the end of round 8. The controller needs to use 3 of these known good processors in round 9 to test the undiagnosed processors in $\mathcal{S} \cup \mathcal{J}$. So there are at least

$$\frac{1}{2}(m+g) - 10$$

good processors that can be used in round 9 to test the fragments. But since at least $1/6$ of the processors in $\mathcal{S} \cup \mathcal{J}$ are faulty we have $g \leq (2/3)m$, so

$$\begin{aligned} \frac{1}{2}(m+g) - 10 &= \frac{1}{2}(m-g) + (g+1) - 11 \\ &\geq \frac{m}{6} + (b+2) - 11 \end{aligned}$$

Since $N > 16$ implies that $m \geq 544$ we have that $(m/6) - 11 \geq 79$. Thus there are sufficient known good processors available in round 9 to test the $b+2$ fragments which suffice to cover the chains.

Finally the controller must diagnose the remaining fragments in $\mathcal{C}_1 \cup \mathcal{C}_2$ in round 10. We claim that this can be done with the known good processors from $\mathcal{S} \cup \mathcal{J}$.

Since the controller has covered every row, it has tested at least one node out of each pair. Wlog it has tested exactly one node from each row. We will say that a row is *good–faulty* if it has tested a good node. We will say that a row is *faulty–good* if it has tested a faulty node, and the row is actually half–faulty. We will say that a row is *faulty–faulty* if it has tested a faulty node and the row is actually all–faulty.

Consider a good–faulty row. Without carrying out any more tests we know the status of the mate of the tested node since each row contains at least one faulty processor. So every fragment that covers a good–faulty row is already diagnosed.

Consider a fragment that the controller has not already diagnosed. Each row the fragment covers is either a faulty–good row or a faulty–faulty row. If the undiagnosed fragment covers a faulty–good row then it must be a good fragment and, as we have seen, there are at most $(b/2)+2$ good fragments. Since there are $b/2$ all-faulty rows there are at most $b/2$ undiagnosed fragments that cover faulty–faulty rows. So there are at most $b+2$ undiagnosed fragments.

We can diagnose these fragments in round 10, using the known good processors from $\mathcal{S} \cup \mathcal{J}$. The argument is identical to the one used above for round 9. In conclusion we have:

Theorem 7 *Complete diagnosis can be performed in ten rounds.*

2.3.5 Constructive algorithm

The previous sections showed that for all n there exists an algorithm which will complete the diagnosis in ten rounds. But the algorithm wasn't explicitly found, since it depended on finding H_{16}. Here we will show that given a finite amount of preprocessing, a suitable test graph can be found in polynomial time for all n.

As in [13], the procedure will be basically the same, only instead of using H_d we shall use the Cayley graphs G_n of Lubotzky, Phillips and Sarnak [41]. As before we will build supernodes and chains, apply the test graph to them, and recover some known good processors from them. These will be used to diagnose the remaining processors.

The problem is that G_n doesn't exist for every value of n. We fix a prime p, with p congruent to 1 modulo 4. Then there is a $(p+1)$-regular graph G_n with $q(q^2-1)/2$ vertices, for any prime q satisfying $(\frac{q}{p}) = 1$, q congruent to 1 modulo 4, where $(\frac{q}{p})$ is the Legendre symbol [10]. The second largest (in absolute value) scaled eigenvalue of this graph is at most $2\sqrt{p}/(p+1)$.

If there is no suitable q define G_n to be the graph $G_{n'}$ along with $n - n'$ isolated vertices, where n' is chosen to be the largest integer smaller than n for which a suitable graph exists.

Lemma 8

$$\lim_{n\to\infty} \frac{n-n'}{n} = 0$$

Proof: Since $(\frac{q}{p})$ depends only on the residue of q modulo p, we can find an arithmetic progression, $a + id$, such that all the primes in the progression are suitable choices for q. It is well known (see eg [48, page 214]) that the density of the primes in an arithmetic progression satisfies

$$\pi_{d,a}(x) \sim \frac{1}{\phi(d)} \frac{x}{\ln x},$$

where $\phi(d)$ is the Euler function. Hence if q_n is the nth prime in the progression, $q_n \sim \phi(d) n \ln n$, which implies

$$\lim_{n\to\infty} \frac{q_{n+1}}{q_n} = 1.$$

The lemma follows. $\square$

So the procedure is as follows. We choose n_0 large enough that for $n > n_0$ the fraction of isolated vertices in G_n is less than ϵ. In this case at least $1/2 - \epsilon$ fraction of the nonisolated vertices must be good nodes.

[13] shows that provided

$$\frac{\sqrt{ab}}{\sqrt{(1-a)(1-b)}} > \frac{2\sqrt{p}}{p+1}$$

then $G_{n'}$ will induce an edge between any sets of vertices A and B where $|A| = an'$ and $|B| = bn'$.

Lemma 2 shows how we can use this condition to induce a large component. A suitable choice of values is to take $\epsilon = 1/30$, and let the extreme values for a and b be $(7/30) + (1/30)$ and $(7/30) - (1/30)$ respectively. (This corresponds to $\lambda = 7/15$ and $\gamma = 1/7$ in the lemma.) Then G_n induces a component with at least $(1/15)n'$ good nodes, provided $p \geq 37$.

Since we need to test both ways, to test the undirected degree-38 graph G_n we will need order-128 supernodes. Complete diagnosis can be performed in $8 + 2 + 2 = 12$ rounds.

If we want to perform diagnosis for a set of supernodes with $n \leq n_0$ we use the nonconstructive algorithm described in Section 2.3.1. The H_{16} graph that this algorithm needs depends on the value of $N \leq \frac{n_0}{32}$. So there are at most $n_0/32$ different H_{16} graphs that could be needed. These graphs can be discovered in advance by doing a finite amount of preprocessing.

2.4 Lower Bounds

In this section we present a lower bound technique. Three-round diagnosis can tolerate $\sqrt{n}/\sqrt{2}$ faults. We will prove that three-round diagnosis cannot tolerate $(8/3)\sqrt{n}$ faults. We know that four-round diagnosis can tolerate $(.03)n$ faults. We will prove that it cannot tolerate $(.42)n$ faults. We have proved that general diagnosis can be performed in ten rounds. The four-round result shows that it requires at least five rounds.

Since three-round diagnosis cannot tolerate $(8/3)\sqrt{n}$ faults, there is no three-round algorithm which can tolerate a linear number of faults. In particular Algorithm 2.1, the four-round algorithm for $t = 0.03n$, is the best possible in the sense that for sufficiently large n there is no three-round algorithm for this problem.

2.4.1 Lower bound technique

We will use an adversary argument. We will view the adversary as deciding which processors are faulty as the testing proceeds.

Initially every processor has an uncertain status. Each time the controller chooses the tests to be performed in some round, the adversary may declare that some processors are in fact faulty. Although the adversary decides which processors to declare faulty in an on-line fashion, the behavior of the processors is indistinguishable from one where the set of faulty processors consists, from the start, of precisely the set of declared faulty processors.

When a declared faulty node is involved in a test, either as tester or testee, the result of the test will always be "faulty". So wlog the controller will not ask for tests involving declared faulty nodes, since it will gain no information by doing so. Tests between nodes of uncertain status always report "good". These tests are worth doing since the adversary may decide to declare that one of the nodes involved in the test is faulty immediately before the tester reports.

The adversary's behavior is limited by the following constraints: It may not declare that t or more processors are faulty, since this would complete the diagnosis. If the adversary declares that a is faulty in some round, and b tested a in an earlier round before a was declared faulty, then the adversary is required to declare that b is also faulty. It may make this declaration in the same round that it declared that a was faulty, or it might have made it in some earlier round. We call this requirement *testing consistency*.

As long as there is some processor b of uncertain status, which has been tested only by declared faulty processors, and which has only tested declared faulty processors then the diagnosis is incomplete. It is impossible for the controller to deduce the status of b. The constraint that fewer than t processors are declared faulty prevents b being diagnosed by a counting argument. The adversary declares processors that b tests to be faulty to ensure that the testing consistency constraint does not force it to declare b to be faulty in some later round.

The algorithms for the adversary given in the next two sections both work by having the adversary declare faulty processors in such a way as to prevent the appearance of large supernodes. In this way the adversary keeps its options open.

Recall from Section 2.1 that T_i denotes the directed graph which has an edge from node a to node b iff processor a tested processor b at some point during the first i rounds and $\overline{T}_i$ denotes the corresponding undirected simple graph. Also G_i and $\overline{G}_i$ are the subgraphs obtained from T_i and $\overline{T}_i$ by restricting the edge set to consist solely of those edges that correspond to good tests.

2.4.2 Lower bound for $t \geq (8/3)\sqrt{n}$

If the adversary uses Algorithm 2.5 then it can prevent diagnosis from being completed in the first three rounds. The next theorem shows that this algorithm can be applied when $t \geq (8/3)\sqrt{n} = 2.67\sqrt{n}$.

Theorem 9 *Complete diagnosis cannot be performed in three rounds if*

$$t \geq (8/3)\sqrt{n} + 1.$$

Proof: We count up the number of faulty processors that the adversary needs to declare to carry out Algorithm 2.5. It declares no faulty processors in round 1.

If $x = 2\sqrt{n}$ then at most

$$2\frac{2n/3}{2\sqrt{n}} = \frac{2}{3}\sqrt{n}$$

faulty vertices are declared in round 2. The biggest component of $\overline{G}_2$ has order $2\sqrt{n} + 1$, so at most

$$\left(\frac{2}{3} + 2\right)\sqrt{n} + 1$$

Input: n processors.

Method:

Round 1 Do not declare any faulty processors.

Round 2 The graph $\overline{T}_2$ of all tests performed has maximum degree 2. So its components are all paths or cycles. Set a parameter x by considering these two cases:

(a) If $\geq \frac{1}{3}n$ nodes are in components of order $\leq 2\sqrt{n}$, then $x = 2\sqrt{n}$.

(b) If $\geq \frac{2}{3}n$ nodes are in components of order $\geq 2\sqrt{n}$, then $x = \sqrt{n}$.

If a component of $\overline{T}_2$ contains more than x vertices provisionally choose as few processors as possible from the component to declare faulty, so that the longest path in $\overline{T}_2$ containing no declared faulty processors contains at most $x - 1$ processors. Let F be the set of chosen processors.

If $a \in F$ and a was tested in round 1, retain testing consistency by replacing a with the processor which tested it. This perturbs each declared faulty processor in $\overline{T}_2$ by possibly moving it to an adjacent vertex. Now the longest path in $\overline{T}_2$ with no faulty processors contains at most $x + 1$ processors.

Ensure that there is at least one processor b, not declared to be faulty, which has either not been tested by the end of round 2, or was tested by a faulty node. If necessary add one more processor to F, to ensure that a suitable b exists.

Round 3 If b participates in a test with some node c that had uncertain status in round 2, then declare that all the processors in the component of $\overline{G}_2$ that includes c are faulty.

Algorithm 2.5: Prevent diagnosis in three rounds

processors are declared faulty in all.

Alternatively if $x = \sqrt{n}$ then at least $2/3$ of the vertices are in components of size at least $2x$, so at most $3/(2x)$ of these vertices are declared faulty. Hence at most

$$2\frac{n/3}{\sqrt{n}} + \frac{3}{2} \cdot \frac{2n/3}{\sqrt{n}} = \left(\frac{2}{3} + 1\right)\sqrt{n}$$

faulty vertices are declared in round 2. The biggest component of $\overline{G}_2$ has order $\sqrt{n}+1$, so at most

$$\left(\frac{2}{3} + 2\right)\sqrt{n} + 1$$

processors are declared faulty in all.

So provided $t \geq (8/3)\sqrt{n} + 1$ the adversary can prevent three-round diagnosis.

□

In fact the $(8/3)\sqrt{n}$ bound is tight up to a constant factor. Let $\tau_3(n)$ denote the largest possible value of t for a given n such that complete diagnosis can be performed in three rounds. Then the following can be shown:

Theorem 10 *The maximum number of faults that can be tolerated for three-round diagnosis to be possible satisfies* $\tau_3(n) = \Theta(\sqrt{n})$.

Proof: We already have that $\tau_3(n) \leq (8/3)\sqrt{n}$. For the converse we will give an algorithm for the controller that completes diagnosis in three rounds.

For **rounds** 1–2 build as many cycles as possible with length $t + 1$ if t is odd, and with length $t + 2$ if t is even. Up to $t + 1$ processors may be left over. Since a cycle cannot consist entirely of faulty nodes, every cycle that is a supernode must be a good supernode.

In **round** 3 use the known good cycles to test the faulty cycles, and any node which isn't in a cycle. This completes the diagnosis.

After round 2 all the processors except possibly $t(t + 2) + (t + 1)$ of them will be diagnosed. So to finish diagnosis in round 3 we need this number to be at most half of the processors. That is we need

$$2t^2 + 6t + 2 \leq n$$

to complete diagnosis in three rounds. □

Theorem 10 shows that 3-round diagnosis is possible if $t \approx \sqrt{n}/\sqrt{2} \approx (0.71)\sqrt{n}$, and Theorem 9 shows that 3-round diagnosis is impossible if $t \approx (8/3)\sqrt{n} \approx (2.67)\sqrt{n}$. It seems likely that a more complex version of Algorithm 2.5, in which more cases and more values of x are considered would reduce the coefficient of $\sqrt{n}$ for the impossibility result. It is possible to show that the technique used in the algorithm cannot obtain a coefficient below 2.27.

2.4.3 In general, diagnosis requires at least five rounds

In this section we shall apply the method of Section 2.4.1 to show that it is impossible to perform complete diagnosis in four rounds. In fact we will show that if $t \geq (0.42)n$ then at least five rounds are needed for complete diagnosis for sufficiently large n.

As with the three-round case the adversary declares some nodes to be faulty in round 2. These are chosen to ensure that the maximum size of a good component in $\overline{G}_2$ is 3 vertices. Let $C_1, C_2, \ldots$ be the components of $\overline{G}_2$ that do not consist of declared faulty nodes.

The adversary then treats round 3 as being a 3-semiregular testing graph T, on the C_i. Since no test was performed between the C_i in rounds 1–2, the adversary is free to declare any individual C_i to be faulty, (but in fact it will only declare subsets of each C_i to be faulty). The adversary will declare faulty nodes in round 3 so as to ensure that $\overline{G}_3$ has a constant bound, independent of n, on the number of vertices in a component.

The following algorithm is used by the adversary to prevent diagnosis from being performed in four rounds. As usual, n is the number of processors. (We will assume that n is even. If n is odd the explanation is made more complex, but the number of faulty processors needed is only affected by an additive constant.)

Round 1 The adversary arbitrarily chooses a processor b. Since the controller cannot possibly gain by leaving processors idle, we shall assume that $\overline{T}_1$ is a complete matching. Let c_1 be the processor that is involved in a test with b.

The adversary declares that c_1 is faulty. No other processor is declared to be faulty. Let F_1 be the set of processors that are declared faulty in this round.

Round 2 Again we assume that every processor is involved in a test in round 2. So each component of $\overline{T}_2$ is either a cycle with an even number of vertices, or two adjacent vertices. The adversary now chooses F_2, the vertices declared faulty in round 2, by considering each component of $\overline{T}_2$ in turn.

If the component contains 2 vertices, then neither vertex is declared faulty. Otherwise the component is a cycle. The adversary arbitrarily selects a vertex that was not tested in round 1 and declares it faulty. With this vertex deleted the remainder of the component is now a path, containing at least three vertices. The adversary works its way along the path, declaring faulty vertices in such a way that it never declares a vertex to be faulty if it was tested in round 1, and at least one vertex out of every four consecutive vertices is declared faulty. Note that it is not possible that three consecutive vertices were all tested in round 1. The exact manner in which the adversary selects vertices to declare faulty is given below. There are several cases to consider, depending on the number of vertices in the path:

$\leq$ **3**: No vertex is declared faulty.

4: The adversary declares exactly one vertex to be faulty.

5: The adversary declares one of the three internal vertices to be faulty. It is impossible that all three of them were tested in round 1.

6: If possible the adversary declares either the third or fourth vertex to be faulty. If both of them were tested in round 1, then neither the second nor fifth vertex could have been tested in round 1. The adversary then declares the both of these vertices to be faulty.

7: If the fourth vertex was not tested in round 1 the adversary declares it to be faulty. On the other hand if it was tested, but neither the third nor fifth vertex was tested, then the adversary declares the third and fifth vertices to be faulty.

If neither of these cases occurs, then the adversary declares the third vertex to be faulty. (It is impossible for all three central vertices to be tested in round 1. So wlog the third vertex was not tested.) Since we are assuming that the fourth and fifth vertex were tested in round 1, the sixth vertex was not tested; the adversary declares it to be faulty as well.

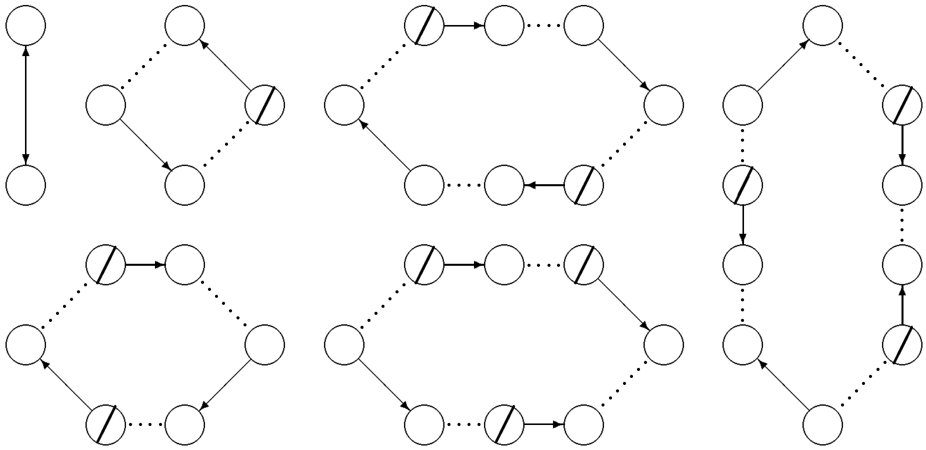


Figure 2.6: Sample arrangements of faulty nodes in a component of T_2. The dotted lines are round 2 tests, the solid lines are round 1 tests. The nodes marked with a slash are declared faulty.

$\geq$ 8: The adversary deals with these cases by declaring a vertex near one end of the path to be faulty. The remainder of the path can then be dealt with by recursion.

If the third vertex from either end in the path was not tested in round 1, then the adversary declares it to be faulty. Otherwise if the third vertices from both ends were tested in round 1, but one of the fourth vertices was not, then the adversary declares that fourth vertex to be faulty.

Otherwise the third and fourth vertices from both ends were tested in round 1. So neither the second or fifth vertex from either end was tested. The adversary arbitrarily chooses an end of the path and declares both the second and fifth vertex from that end to be faulty.

Let E_2 be the set of processors chosen above to be declared faulty. Some example choices of vertices in E_2 which satisfy this construction are shown in Figure 2.6.

However the actual set F_2 of processors declared faulty may be a little different from E_2. Let c_2 be the processor that participated in a test with b in round 2, and let c_2' be the processor that participated in a test with c_2 in round 1. Then define

$$F_2 = (E_2 \cup \{c_2, c_2'\}) \setminus \{b\}.$$

So $|F_2| \leq |E_2| + 2$.

Since a test reports good iff neither vertex that participates in the test is declared faulty, the components of $\overline{G}_2$ are identical to the components of $\overline{T}_2 \setminus F_2$. Let $C_1^1, C_2^1, \ldots, C_{n_1}^1$ be the components of $\overline{G}_2$ which contain only 1 vertex, $C_1^2, C_2^2, \ldots, C_{n_2}^2$ be the components with 2 vertices and $C_1^3, C_2^3, \ldots, C_{n_3}^3$ be the components with 3 vertices. Observe that no component will contain more than three vertices.

Round 3 The adversary constructs an undirected graph $\overline{G}$, on the C_i^j, by connecting $C_{i_1}^{j_1}$ to $C_{i_2}^{j_2}$ iff there is a test between them in T_3. So $\overline{G}$ is 3-semiregular graph.

Fix an integer r. Corollary 26 (which will be proved in Section 2.5.3) states that the adversary can select a set W from the vertices of $\overline{G}$ where $|W| \leq \frac{1}{4}n_3 + O(\frac{1}{\sqrt{r}}n)$, such that deleting W separates $\overline{G}$ into components of size at most r.

By Corollary 26, W does not contain any vertices of degree 1 in $\overline{G}$. So it contains no vertices corresponding to C_i^1 components. If in round 3, one node in a C_i^2 or C_i^3 component tests another node in the same component, then the corresponding vertex in $\overline{G}$ has degree at most 1, and so it is also not in W.

The adversary now selects E_3, the tentative set of processors it declares to be faulty in round 3. Its aim is to select processors that correspond to the vertices in W. It does not declare all these processors to be faulty, since then $|F_2 \cup E_3|$ might be greater than $n/2$.

If C_i^2 is in W, then the adversary declares both processors in C_i^2 to be faulty. If $C_i^3 \in W$ then two processors from C_i^3 are declared faulty. Since C_i^3 isn't a 3-cycle, there is a processor in C_i^3 that has not tested either of the other two processors. So declaring the other two processors to be faulty will not violate the consistency requirement. So for each C_i^3 component in W, two processors are added to E_3.

Let c_3 be the processor that participated in a test with b in round 4. To ensure that b is not diagnosed the adversary declares that the component C_* of $\overline{G}_2$ that includes c_3 is faulty. But b is not declared to be faulty. So the set F_3 of processors that are declared faulty in round 3 is given by

$$F_3 = (E_3 \cup C_*) \setminus \{b\}.$$

So $|F_3| \leq |E_3| + 3$.

Round 4 If b participates in a test with some processor c_4, then the adversary declares c_4 to be faulty. If $c_4 \notin F_1 \cup F_2 \cup F_3$ then the adversary maintains consistency by declaring the entire component of $\overline{G}_3$ that includes c_4 to be faulty. As we shall see, this is a bounded number of processors. Let F_4 be the set of processors declared faulty in round 4.

The controller has been prevented from diagnosing b, and so diagnosis is still incomplete.

The following lemma is used to to show the relationship between $|E_2|$ and $|E_3|$:

Lemma 11 *Let n_3 be the number of components of $\overline{T}_2$ that have order exactly 3. Then the following inequality holds:*

$$|E_2| + \frac{1}{2}n_3 \leq \frac{5}{12}n.$$

Proof: Consider a path containing i vertices. Suppose that vertices in the path are declared to be faulty using the strategy given for round 2 above. We charge 1 for each vertex that is declared faulty, and $\frac{1}{2}$ for each component of order-3 that remains after deleting the declared faulty vertices. Let $\xi(i)$ be the largest possible charge for a path of i vertices, assuming that the adversary follows the given strategy.

The values of $\xi(1)$ to $\xi(7)$ are obtained by directly examining the strategy. We obtain the following a recurrence equation for $\xi(i)$ when $i \geq 8$.

$$\xi(i) \leq \max\left\{1 + \xi(i-3),\ \frac{3}{2} + \xi(i-4),\ 2 + \xi(i-5)\right\} \tag{2.7}$$

Suppose that both the third and fourth processors in a path were tested in round 1. Then the second processor must have tested the third processor, and the fifth processor must have tested the fourth processor. Since the round 1 tests take place between consecutive pairs, each round 1 test took place between processor $2j$ and processor $2j+1$ for some j. Then if i is odd, there must be a round 1 test between the fourth from last and the third from last processor. Since the adversary's strategy only declares the fifth vertex to be faulty in situations where it cannot declare the third or fourth vertex from *either* end to be faulty, it doesn't ever declare the fifth vertex to be faulty when i is odd. So when i is odd, and $i \geq 8$ we can improve on (2.7) to obtain

$$\xi(i) \leq \max\left\{1 + \xi(i-3),\ \frac{3}{2} + \xi(i-4)\right\} \tag{2.8}$$

Using (2.7) and (2.8) we can tabulate some upper bounds for $\xi(i)$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\xi(\mathbf{i})$	0	0	$\frac{1}{2}$	$\frac{3}{2}$	$\frac{3}{2}$	2	2	3	3	$\frac{7}{2}$	4	$\frac{9}{2}$	$\frac{9}{2}$	5	$\frac{11}{2}$

For $i \geq 16$ we only need to use (2.7). If we select any range of 5 consecutive values, we can repeatedly apply the recurrence equation until we express $\xi(i)$ in terms of a $\xi(j)$ in that range. In particular we have for $i \geq 16$

$$\xi(i) \leq \max_{9 \leq j \leq 13}\left\{\xi(j) + \frac{2}{5}(i-j)\right\} = \xi(12) + \frac{2}{5}(i-12) = \frac{2}{5}i - \frac{3}{10} \tag{2.9}$$

The cost of a component of $\overline{T}_2$ is calculated by charging one for each vertex declared faulty, and $\frac{1}{2}$ for each component of 3 vertices that remains after the faulty vertices are deleted. So if the component is a pair of adjacent vertices, it costs nothing. If it is a cycle of length $2d$, then the cost is at most $1 + \xi(2d-1)$. But for all $d \geq 2$ we have using the table and equation (2.9) that

$$1 + \xi(2d-1) \leq \frac{5}{12}2d. \tag{2.10}$$

The result follows. □

The analysis in the lemma is not tight for cycles of length 12. A better result can be obtained by being more specific about which processor in the cycle should first be declared faulty. However the $\frac{5}{12}$ fraction used in the lemma cannot be improved. Equality is obtained in (2.10) for cycles of 6 vertices. See the 6-cycle in figure 2.6.

We also need a bound on $|F_4|$.

Lemma 12

$$|F_4| \leq 4r + 2$$

Proof: If b does not participate in a test with a processor c_4 in round 4 or if c_4 was declared faulty in an earlier round, then $F_4 = \emptyset$. Otherwise let Γ be the component of $\overline{G}_3$ which includes c_4.

Let $\overline{G}$ be the graph defined in the construction for round 3. Then Γ corresponds to a component of $\overline{G} \setminus W$ which has order at most r. Each node in $\overline{G}$ corresponds to a set of at most 3 nodes in $\overline{T}_3$. Let Γ' be the union of these sets.

Any node of $\overline{T}_3$ that is adjacent to Γ' and which was not declared faulty in one of the first two rounds must belong to a component C_i^j that corresponds to a node of W. In most cases the vertices of such components were declared faulty in round 3. However only two out of the three vertices in a C_i^3 component were declared faulty. So each C_i^3 component adjacent to Γ' in $\overline{T}_3$ may contribute an additional vertex to Γ.

Since $\overline{G}$ is 3-semiregular, a connected subgraph of order at most r may have at most $r+2$ neighbors in $\overline{G}$ that are not in the subgraph. So there are at most $r+2$ of the C_i^3 components adjacent to Γ' that could contribute an extra node. Hence

$$|F_4| \leq 3r + (r+2) = 4r + 2 \qquad \square$$

Using the last two lemmas we are now in a position to prove the main theorem of this section:

Theorem 13 *There exists a constant c such that complete diagnosis of n processors* *is impossible in less than five rounds if*

$$t \geq \frac{5}{12}n + cn^{\frac{2}{3}}.$$

Proof: From the construction, and Lemma 12 we have the following upper bounds on the number of declared faulty processors in each round.

$$\begin{aligned} |F_1| &\leq 1 \\ |F_2| &\leq |E_2| + 2 \\ |F_3| &\leq |E_3| + 3 \\ |F_4| &\leq 4r + 2 \end{aligned} \tag{2.11}$$

By Corollary 26 in section 2.5.3 we have

$$|E_3| \leq \frac{1}{2}n_3 + O(\frac{1}{\sqrt{r}}n)$$

where n_3 is the number of components, of order 3 that are created in round 2. Applying Lemma 11 we get

$$|E_2| + |E_3| \leq \left(|E_2| + \frac{1}{2}n_3\right) + O(\frac{1}{\sqrt{r}}n) \leq \frac{5}{12}n + O(\frac{1}{\sqrt{r}}n). \tag{2.12}$$

Let $r = n^{2/3}$, so $n/\sqrt{r} = n^{2/3}$. Summing the bounds (2.11) on the F_i and using (2.12) we obtain

$$|F_1 \cup F_2 \cup F_3 \cup F_4| \leq \frac{5}{12}n + O(n^{\frac{2}{3}}). \tag{2.13}$$

The theorem follows immediately from (2.13). □

Thus our five round lower bound holds when 42% or more of the vertices are faulty, for sufficiently large n.

2.5 Graph Separators

In this section we study separators of 3-semiregular graphs. We will prove that any 3-semiregular graph can be separated into components of order at most r, by deleting at most $(1/4 + O(1/r))n$ vertices from the graph. We will also show by example that the fraction 1/4 is the best possible.

2.5.1 Definitions

Definition 14 Let $G = (V, E)$ be a graph and let Γ be a set of graphs. $S \subseteq V$ is a (λ, Γ)-*separator* of G if $|S| \leq \lambda|V|$ and every component of $G \setminus S$ belongs to Γ.

Define K_i to be the complete graph on i vertices. In particular K_1 is the graph with a single vertex. Let C_i be a cycle of length i. Let Γ_r be the set of all graphs of order r or less.

A *3-semiregular graph* G is an undirected graph in which every vertex has degree at most 3. It need not contain any degree 3 vertices.

Let $G = (V, E)$. Let w be a new vertex, i.e. $w \notin V$. An *edge bisection* of G is any graph of the form $(V \cup \{w\}, (E \setminus \{uv\}) \cup \{uw, wv\})$ where $u, v \in V$ and $uv \in E$.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where V_1 and V_2 are disjoint. Then a *concatenation* of G_1 and G_2 is any graph of the form $(V_1 \cup V_2, E_1 \cup E_2 \cup \{v_1v_2\})$ where $v_1 \in V_1$ and $v_2 \in V_2$.

Our graph separation theorem is based on several lemmas that separate a 3-semiregular graph into components that have a simpler structure. We will make use of the classes of graphs $\Upsilon(\mathcal{A}, \mathcal{B})$ defined below.

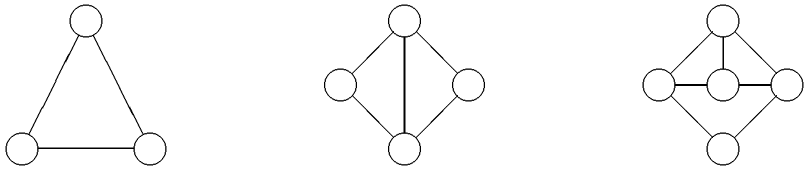
Definition 15 Let $\mathcal{A}$ and $\mathcal{B}$ be sets of graphs. We define a set of graphs, $T(\mathcal{A}, \mathcal{B})$, to be the closure of $\mathcal{B} \cup \{K_1\}$ under edge bisection and also under concatenation with elements of $\mathcal{A} \cup \{K_1\}$.

We define $\Upsilon(\mathcal{A}, \mathcal{B})$ to be the set of 3-semiregular graphs in $T(\mathcal{A}, \mathcal{B})$.

We call $\mathcal{B}$ the set of *base graphs* in $\Upsilon(\mathcal{A}, \mathcal{B})$.

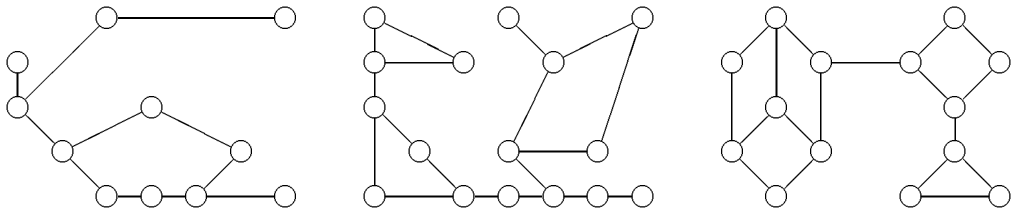
We call $\mathcal{A}$ the set of *disjoint graphs* in $\Upsilon(\mathcal{A}, \mathcal{B})$.

For example $\Upsilon(\emptyset, \emptyset)$ is the set of 3-semiregular trees, and $\Upsilon(\emptyset, \{C_3\})$ is the set of all 3-semiregular graphs that contain at most one cycle. Some further examples are given in Figure 2.8, where ○, ⦶ and ⊖ denote the graphs shown in Figure 2.7.



$\bigcirc$ graph $\equiv C_3$ $\quad$ Φ graph $\quad$ $\ominus$ graph

Figure 2.7: Some graph definitions



$\Upsilon(\emptyset, \{\bigcirc\})$ graph — At most one cycle $\quad$ $\Upsilon(\{\bigcirc\}, \{\bigcirc\})$ graph — Several disjoint cycles $\quad$ $\Upsilon(\{\bigcirc\}, \{\bigcirc, \ominus\})$ graph — Many cycles, one $\ominus$

Figure 2.8: Some examples of $\Upsilon(,)$-type graphs

2.5.2 Separation lemmas

Lemma 16 *Let* $\Gamma = \Upsilon(\emptyset, \{\bigcirc, \Phi\})$. *Then every 3-semiregular graph* G *has a* $(1/3, \Gamma)$-*separator.*

Proof: We use induction on the order of G.

Let $G = (V, E)$ be a 3-semiregular graph of order n, and suppose the lemma has been shown for all graphs of order less than n. We will show that G has a $(1/3, \Gamma)$-separator.

If G is disconnected, then let G' be a component of G. Since $|G'| < n$, there exists by induction a $(1/3, \Gamma)$-separator, S' of G'. Similarly there exists a $(1/3, \Gamma)$-separator, S of $G \setminus G'$. So $S' \cup S$ is a $(1/3, \Gamma)$-separator of G. Clearly if $G \in \Upsilon(\emptyset, \{\bigcirc, \Phi\})$ then the lemma is trivially satisfied.

Suppose that G contains a vertex a with degree 1. Let $G' = G \setminus \{a\}$. By induction there is a $(1/3, \Gamma)$-separator, S' of G'. Since the graphs in Γ are closed under concatenation with the graph K_1, S' must be a $(1/3, \Gamma)$-separator of G as well.

Thus we have a reduction from G to G' that can be applied to any graph that contains a vertex of degree 1. For other 3-semiregular graphs we will introduce two similar reductions that can be used to obtain smaller graphs G' which can also be separated by the inductive hypothesis.

First suppose that we have a situation like that shown in Figure 2.9. We call this a *base reduction*.

In Figure 2.9(a) G is reduced to G' by removing the two vertices a and b with a slash drawn through them (which will become part of the separator S), and the vertices that have a ring drawn around them (which will be a component of $G \setminus S$). The remainder of the graph is called G'. Since it is a subgraph of G it is a 3-semiregular graph. So by induction, G' has a $(1/3, \Gamma)$-separator, S'.

Let $S = S' \cup \{a, b\}$. Then $|S| \leq \frac{1}{3}n$ since S includes at most $1/3$ of the vertices that were removed from G to obtain G'. The component formed by the ringed vertices in $G \setminus S$ is a Φ graph, so it belongs to Γ. Therefore S is a $(1/3, \Gamma)$-separator of G. Figure 2.9(b) shows $G \setminus S$.

We can use any similar base reduction to reduce G to G'. A suitable reduction needs the properties:

- The number of deleted vertices (i.e. vertices with a slash in the diagram) must be at most one third of the number of vertices in $G \setminus G'$.

- The component that is separated from the main body of G must be in Γ. This will certainly be true if it is a cycle or Φ.

Appendix A gives a table of 32 graph reductions. The reductions numbered 1, 2, 5, 6, 7, 9, 13, 21 and 23 are all base reductions that can be used to prove the inductive step in this lemma.

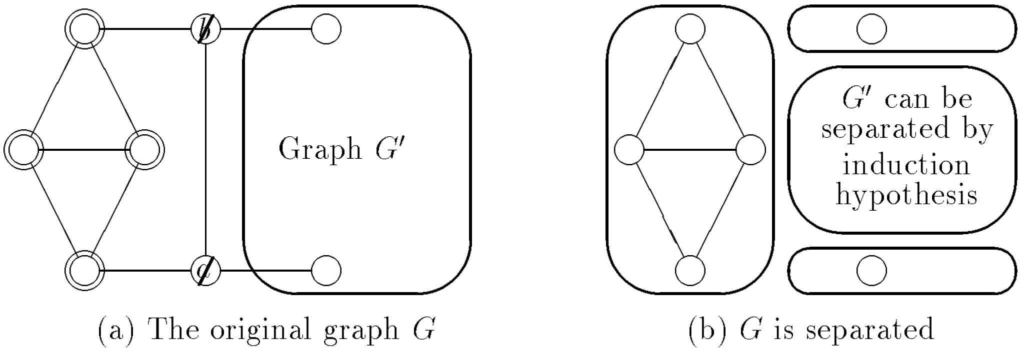


Figure 2.9: Example of a base reduction. A separator S' for G' is obtained inductively. $S = S' \cup \{a, b\}$ is a separator for G. The ringed vertices form a component of $G \setminus S$. Note that the two vertices shown in G' could turn out to be in the same component of $G' \setminus S'$, or they could be in different components (as shown) or one or both of them could be deleted.

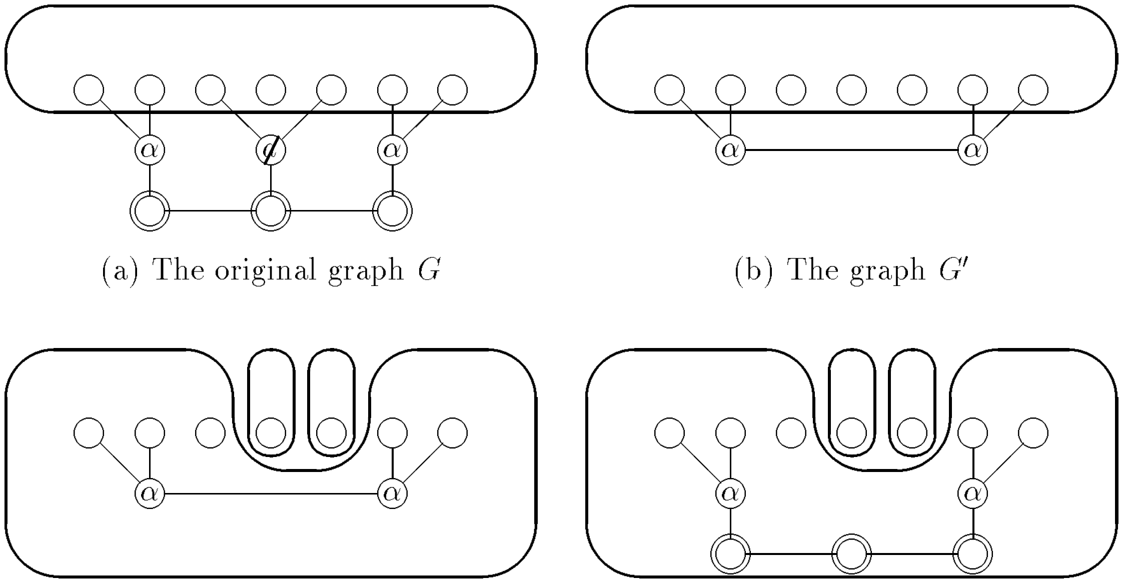
However base reductions are not sufficient to cover all the possible choices of G. We will also consider a second type of reduction called a *bridge reduction*. An example of a bridge reduction is shown in Figure 2.10.

Figure 2.10(a) shows the original graph G. It is reduced to G' which is shown in part (b) of the figure. The vertex, a, with a slash through it and the ringed vertices are deleted. A new edge is placed between the two vertices in G that are marked with the an α. These are the vertices that are adjacent to the ringed vertices. Since both α-vertices had an edge removed from them, adding the new edge will not violate the restriction that each vertex has degree at most 3.

So by induction G' has a $(1/3, \Gamma)$-separator, S'. Figure 2.10(c) shows $G' \setminus S'$. We take $S = S' \cup \{a\}$. Figure 2.10(d) shows $G \setminus S$. Then $|S| \leq \frac{1}{3}n$ because S includes at most 1/3 of the vertices that were removed from G to obtain G'.

Suppose that both of the α vertices belong to S'. Then the component containing the ringed vertices is a path, and is thus in Γ.

If exactly one of the α vertices is in S', then the component of $G \setminus S$ that contains the ringed vertices can be constructed from a component of $G' \setminus S'$ by performing



(a) The original graph G (b) The graph G'

(c) G' is separated (d) G is separated

Figure 2.10: Example of a bridge reduction. G' is obtained by deleting the ringed vertices and the vertex marked with a slash and by inserting a new edge between the vertices labeled with an α. A separator S' for G' is obtained inductively. $S = S' \cup \{a\}$ is a separator for G. Note that it is possible that one or both of the α vertices are deleted from G' when it is separated.

several concatenations with the K_1 graph. Since $\Gamma = \Upsilon(\emptyset, \{\bigcirc, \Phi\})$ is closed under concatenations with K_1 the component of $G \setminus S$ is also in Γ.

Finally if neither of the α vertices is in S', then the component of $G \setminus S$ that contains the ringed vertices can be constructed from a component of $G' \setminus S'$ by performing several edge bisections. Since $\Gamma = \Upsilon(\emptyset, \{\bigcirc, \Phi\})$ is closed under edge bisections the component of $G \setminus S$ is also in Γ.

Thus we have shown that S is a $(1/3, \Gamma)$-separator of G. We can use any similar bridge reduction to reduce G. A suitable reduction needs the properties:

- The number of deleted vertices (i.e. vertices with a slash in the diagram) must be at most one third of the number of vertices in $G \setminus G'$.

- The two end vertices of the bridge (the α-vertices in the diagram) must not be adjacent in G, so that an edge can be inserted between them in G'.
- It is possible for a reduction to introduce several bridges simultaneously. In that case there must not be two bridges that share the same pair of end points. But no problem is caused if two different bridges have one end point in common. (In fact there are several examples of this in appendix A; e.g., reduction 16)

The reductions numbered 4, 11, 15, 16, 17, 18, 19, 25, 26, 28, 29, 31 and 32 are all bridge reductions that can be used to prove the inductive step in this lemma.

To complete the proof of the lemma we need to show that whenever $G \notin \Gamma$ then, either one of the base reductions or one of the bridge reductions, can be applied to G. In fact we will show that a reduction can be applied unless G is a tree or $G \in \{\bigcirc, \obar\}$.

Suppose G is not a tree. Then G contains a cycle $M \subseteq V$.

Let $D \subseteq V$ be the set of vertices adjacent to M but not in M. Let A be the set of vertices adjacent to D in $V \setminus (M \cup D)$. Let $B = V \setminus (M \cup D \cup A)$, the remainder of the vertices of G. See Figure 2.11 for an example of this nomenclature.

We take cases depending on the quantities $|M|$, $|D|$ and $|A|$. The reductions listed in appendix A are in lexicographic order on these quantities. In some cases it is also necessary to consider subcases based on the number of edges between D-vertices and on some other quantities.

For example, the graph in Figure 2.11 shows a case in which $|M| = |D| = 4$ and $|A| = 6$. Reduction 29 can be applied to any graph with $|M| = |D| = 4$. The vertex marked with a slash in Figure 2.11 should be deleted. The three ringed vertices will be removed (using bridges) to construct G'. The three extra edges that would have to be added to G' have been shown as dotted lines in the figure.

If a combination of $|M|$, $|D|$ and $|A|$ values is not shown in the appendix it means that no 3-semiregular graph exists with these values. For example $|A| \leq 2|D|$ since each D-vertex can be connected to at most two A-vertices. Those reductions listed in the appendix as *disjoint reductions* are not used to prove this lemma, and for now

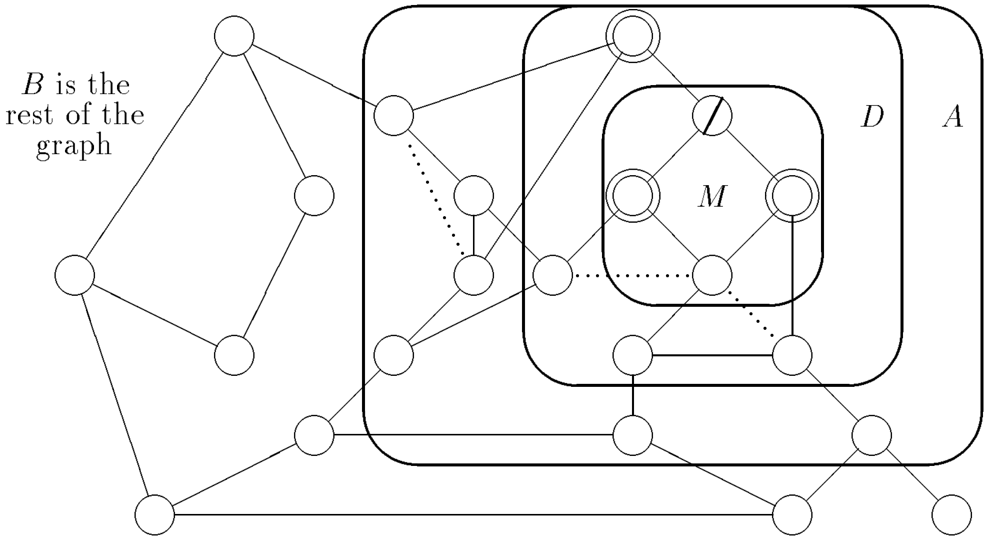


Figure 2.11: Example of M, D, A, and B

should be ignored.

We assume that the reductions in the appendix are applied in the order that they are listed there. That is that a later reduction is applied only if none of the earlier reductions were applicable. For example when considering reductions with $|M| = 4$ we may assume that the graph contains no triangles.

Examining the appendix shows that there is a reduction for every possible combination of $|M|$, $|D|$ and $|A|$ values. Hence the induction carries through and the lemma follows. $\square$

Next we extend Lemma 16 to obtain a $(1/4, \Gamma)$-separator for 3-semiregular graphs, with $\Gamma = \Upsilon(\{\bigcirc, \obar\}, \mathcal{B})$ where $\mathcal{B}$ is a set of graphs that will be defined below. We use essentially the same method as was used to prove Lemma 16. However we can no longer use any reduction whose ratio of deleted vertices to $|G \setminus G'|$ is greater than $\frac{1}{4}$. In order to obtain reductions with ratio $\frac{1}{4}$ or better we consider a new class of reductions, called *disjoint reductions*.

Figure 2.12(a) shows the original graph G. It is reduced to G' which is shown in part (b) of the figure. The vertex, a, with a slash through it and the ringed vertices are deleted. The remaining graph, G', is a 3-semiregular graph.

So by induction G' has a $(1/4, \Gamma)$-separator, S'. Figure 2.12(c) shows $G' \setminus S'$. We take $S = S' \cup \{a\}$. Figure 2.12(d) shows $G \setminus S$. Then $|S| \leq \frac{1}{3}n$ because S includes at most 1/4 of the vertices that were removed from G to obtain G'.

Suppose that the α vertex belong to S'. Then the component of $G \setminus S$ that includes the ringed vertices is the $\bigcirc$ graph. Thus it is in Γ.

Otherwise if the α vertex is not in S', then the component of $G \setminus S$ that contains the ringed vertices can be constructed from a component of $G' \setminus S'$ by performing a concatenation with with the $\bigcirc$ graph. Since $\Gamma = \Upsilon(\{\bigcirc, \obar\}, \mathcal{B})$ is closed under concatenations with $\bigcirc$ the component of $G \setminus S$ is also in Γ.

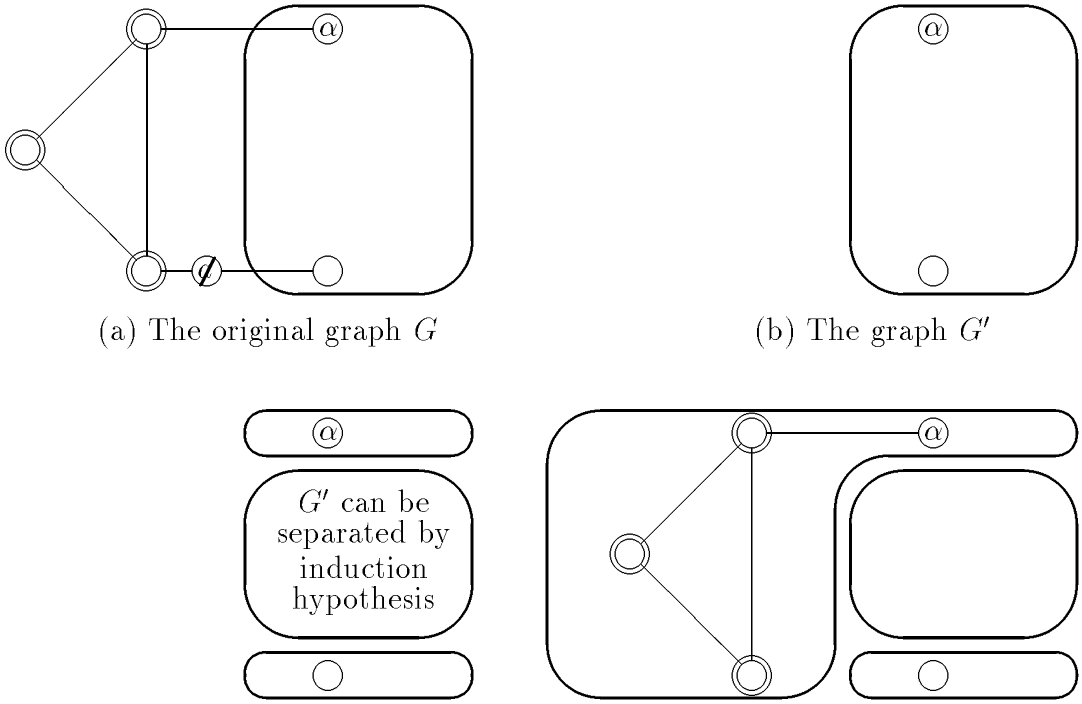
Thus we have shown that S is a $(1/4, \Gamma)$-separator of G. We can use any similar disjoint reduction to reduce G. A suitable reduction needs the properties:

- The number of deleted vertices is at most one quarter of the vertices in $G \setminus G'$.
- The graph that is concatenated with a component of $G' \setminus S'$ is in $\{\bigcirc, \obar\}$.

Appendix A lists the following applicable disjoint reductions: reductions numbered 3, 14, 22, 24 and 27.

In order to apply induction we need to show that any graph not in $\Upsilon(\{\bigcirc, \obar\}, \mathcal{B})$ has a suitable reduction. We can use the same M, D and A classification that was used in the proof of Lemma 16. Any of the reductions listed in the Appendix that has ratio 1/4 or less is suitable. However, some of the reductions that were used in Lemma 16 are no longer suitable because they have ratio $> 1/4$. For example reduction 9 is no longer suitable because its ratio is $\frac{2}{7} > \frac{1}{4}$. Note that some of the new reductions require the use of $\ominus$ as an additional base graph.

Definition 17 We will say that a 3-semiregular graph G is a Δ-*graph* if every vertex of G lies in exactly one 3-cycle.



(a) The original graph G (b) The graph G'

(c) G' is separated (d) G is separated

Figure 2.12: Example of a disjoint reduction. G' is obtained by deleting the ringed vertices and the vertex marked with a slash. A separator S' for G' is obtained inductively. $S = S' \cup \{a\}$ is a separator for G. Note that it is possible that the vertex marked with an α belongs to S'.

Figure 2.13 shows an example of a Δ-graph. Examining the appendix shows that there is a suitable reduction for every 3-semiregular graph, G, except when G is a tree (in which case no reduction is needed) or when G is a Δ-graph.

Lemma 18 *Let G be a connected 3-semiregular graph which is not a tree and which cannot be reduced by any of the $1/4$ ratio reductions given in Appendix A. Then G is a Δ-graph.*

Proof: Suppose that G cannot be reduced by any of the ratio $\frac{1}{4}$ reductions in the Appendix. There is only one bad case namely "reduction" 20. This case consists of a 3 cycle, each of whose vertices is adjacent to a distinct, disjoint 3-cycle. (There doesn't seem to be a simple reduction which covers this case.)

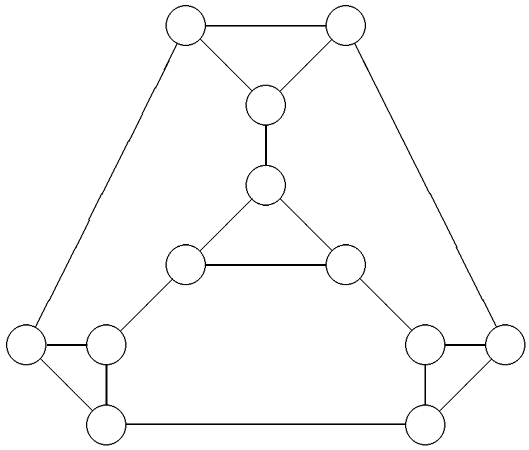


Figure 2.13: An example of a Δ-graph

If G contained a 3-cycle whose vertices are not adjacent to disjoint, distinct 3-cycles then we could apply one of the reductions to G. Similarly we could reduce G if it had no 3-cycles at all. Thus the only graphs that cannot be reduced are Δ-graphs.

$\square$

Definition 19 Let Λ be the set of all graphs G such that G can be reduced to a $\Upsilon(\{\bigcirc\}, \emptyset)$ graph by deleting at most two vertices from it.

Lemma 20 *Every Δ-graph has a $(\frac{2}{9}, \Lambda)$-separator.*

Proof: Suppose G is a Δ-graph and define an auxiliary graph H by placing a node in H for every 3-cycle in G. Connect two nodes in H iff the corresponding cycles in G are adjacent. (In other words H is the graph obtained from G by performing a delta–star transformation on every three cycle in G.) H must be a 3-regular graph.

By Lemma 16, H has a $(1/3, \Upsilon(\emptyset, \{\bigcirc, \oplus\}))$-separator S_H. Define a separator S_G for G as follows: for each node in S_H, let S_G contain 2 of the vertices from the corresponding 3-cycle in G. S_G separates G into components corresponding to the components of $H \setminus S_H$ and $|S_G| = 2|S_H| \leq 2 \cdot \frac{1}{3}|H| = \frac{2}{9}|G|$.

Let C_H be a component of $H \setminus S_H$ and let C_G be the corresponding component of $G \setminus S_G$. If C_H is a tree then $C_G \in \Upsilon(\{\bigcirc\}, \emptyset)$ because the only cycles in C_G are 3-cycles; all of which are disjoint.

Otherwise C_H contains $\bigcirc$ or $\varobar$ as a base graph. Since C_H may contain only one base graph, we may obtain a tree by deleting a single node from C_H. This corresponds to deleting two vertices from the C_G component to obtain a graph in $\Upsilon(\{\bigcirc\}, \emptyset)$. Thus each component of $G \setminus S_G$ is in Λ. $\square$

Lemma 21 *Let G be a 3-semiregular graph. Let Γ be $\Upsilon(\{\bigcirc, \varobar\}, \{\bigcirc, \varobar, \ominus\} \cup \Lambda)$. Then G has a $(\frac{1}{4}, \Gamma)$-separator S.*

Proof: By induction on $|G|$. If $G \in \Gamma$ then we take $S = \emptyset$. Otherwise, if G is not a Δ-graph then by Lemma 18 it can be reduced, by a reduction from the Appendix, to a graph G' to which we apply the inductive hypothesis. If G is a Δ-graph then by Lemma 20 it has a $(\frac{2}{9}, \Lambda)$-separator, and so we are done. $\square$

2.5.3 $(1/4 + O(1/r), \Gamma_r)$-separators of 3-semiregular graphs

Our objective is to obtain a separator of 3-semiregular graphs into components that contain r vertices or less. The components that result from Lemma 21 can be arbitrarily large, but they are simple enough that they can be reduced to bounded size components by deleting $O(\frac{n}{r})$ additional vertices. The following lemma shows how this is done for trees. Recall that we defined Γ_r to be the set of graphs of order r or fewer.

Lemma 22 (folklore) *Let G be a tree with n vertices, and let r be some positive integer. Then G has a $(\frac{1}{r}, \Gamma_{r-1})$-separator.*

Proof: The lemma is clearly true if $n < r$. If $n \geq r$ assume that the lemma hold for all trees with fewer than n vertices. We construct a suitable separator.

Choose an arbitrary root v for G. Let $G(u)$ denote the subtree of G rooted at u. An easy induction shows that there is a node p such that $|G(p)| \geq r$ and for all children c of p, $|G(c)| < r$. By the inductive hypothesis, the tree $G \setminus G(p)$ has a $(\frac{1}{r}, \Gamma_{r-1})$-separator S. Then $S \cup \{p\}$ is a suitable separator for G. $\square$

Lemma 23 *Let $G \in \Upsilon(\{\bigcirc\}, \emptyset)$ be an order-n graph, and let r be some positive integer. Then G has a $(\frac{2}{r}, \Gamma_{r-1})$-separator.*

Proof: The lemma is clearly true if $n < r$. If $n \geq r$ assume that the lemma holds for all $\Upsilon(\{\bigcirc\}, \emptyset)$ graphs with fewer than n vertices. We construct a suitable separator.

Construct an auxiliary graph H by taking a vertex in H for every cycle in G and for every vertex in G that is not in any cycle. For each pair of adjacent vertices in G that correspond to distinct vertices in H, connect the corresponding vertices in H. Then H is a tree.

If H consists of a single node and $n \geq r$ then G is a cycle. Since the lemma is clearly true for cycles we may assume that H contains at least 2 vertices.

Choose a root v for H where v is a leaf vertex of H. For u a vertex of H let $H(u)$ denote the subtree of H rooted at u and let $G(u)$ be the subgraph of G that corresponds to the vertices in $H(u)$. An easy induction shows that there is a node p in H such that $|G(p)| \geq r$ and for all children c of p in H, $|G(c)| < r$. Let q be a vertex of G chosen as follows:

- If p corresponds to a single vertex in G then set q to that vertex.
- If $p \neq v$ and p corresponds to a cycle C in G, then let p' be the parent of p in H. By construction there will be exactly one node in C that is adjacent to a node of G that corresponds to p'. Set q to that node in C.
- If $p = v$ and p corresponds to a cycle C in G, then set q to the unique node in C that is adjacent to a node of $G \setminus C$. This node is unique since v is a leaf vertex of H.

The graph $G \setminus \{q\}$ has 2 components. One component is a $\Upsilon(\{\bigcirc\}, \emptyset)$ graph with fewer than n vertices. By induction it has a $(\frac{2}{r}, \Gamma_{r-1})$-separator S. The other component is a tree which contains at least $r-1$ vertices. By Lemma 22 it has a $(\frac{1}{r}, \Gamma_{r-1})$-separator S'. Then $S \cup S' \cup \{q\}$ is a $(\frac{2}{r}, \Gamma_{r-1})$-separator for G. □

Lemma 24 *Let $G \in \Upsilon(\{\bigcirc, \obar\}, \emptyset)$ be an order n graph, and let r be some positive integer. Then G has a $(\frac{3}{r}, \Gamma_{r-1})$-separator.*

Proof: By a similar argument to that used for Lemma 23. □

Theorem 25 *Let $r > 1$. Any 3-semiregular graph G has a $(\frac{1}{4}+O(\frac{1}{r}), \Gamma_{r-1})$-separator.*

Proof: By Lemma 21, G has a $(\frac{1}{4}, \Upsilon(\{\bigcirc, \obar\}, \{\bigcirc, \obar, \ominus\} \cup \Lambda))$-separator S. Define a $(\frac{2}{r}, \Upsilon(\{\bigcirc, \obar\}, \emptyset) \cup \Gamma_{r-1})$-separator S_C for each component C of $G \setminus S$ as follows:

- If $|C| < r$ or C is a $\Upsilon(\{\bigcirc, \obar\}, \emptyset)$ graph then $S_C = \emptyset$.
- If C contains a $\ominus$ base graph, then we may obtain a $\Upsilon(\{\bigcirc, \obar\}, \emptyset)$ graph from C by deleting a single vertex a. Let $S_C = \{a\}$.
- If C contains a Λ base graph, then by Definition 19, we may obtain a $\Upsilon(\{\bigcirc, \obar\}, \emptyset)$ graph from C by deleting two vertices a and b. Let $S_C = \{a, b\}$.

By Lemma 24, each component C' of $G \setminus (S \cup \bigcup_C S_C)$ has a $(\frac{3}{r}, \Gamma_{r-1})$-separator $S'_{C'}$. Then

$$S \cup \bigcup_C S_C \cup \bigcup_{C'} S'_{C'}$$

is a $(\frac{1}{4} + \frac{5}{r}, \Gamma_{r-1})$-separator of G. □

Theorem 25 is the main theorem of this section. However for the application we need to show that only a $\frac{1}{4}$ fraction of the degree-3 vertices need be deleted, instead of a $\frac{1}{4}$ fraction of all the vertices.

Corollary 26 *Let G be a 3-semiregular graph with order n. Let n_1, n_2, and n_3 be the number of vertices in G with degrees 1, 2, and 3 respectively. Let $r \geq 4$. Then G can be separated into components of size at most r by deleting at most*

$$\frac{n_3}{4} + O\left(\frac{n_3}{\sqrt{r}}\right) \text{ degree-3 vertices and } O\left(\frac{n_2}{\sqrt{r}}\right) \text{ degree-2 vertices.}$$

No degree 1 vertex is deleted.

Proof: Construct an auxiliary graph H from G by placing a node in H for each degree-3 vertex in G. Connect two vertices u and v in H iff there is a path $ux_1x_2\ldots x_kv$ in G such that each x_i vertex has degree-2. Then H is a 3-semiregular graph of order n_3. By Theorem 25, H has a $(\frac{1}{4} + O(\frac{1}{\sqrt{r}}), \Gamma_{\sqrt{r}})$-separator, S_H.

Now we will construct a separator for G. For $P = x_1x_2\ldots x_k$, a maximal path of degree-2 vertices in G, let S_P be $\emptyset$ if $k < \frac{1}{5}\sqrt{r}$. Otherwise let S_P be $\{x_1, x_{1+(1/5)\sqrt{r}}, x_{1+(2/5)\sqrt{r}}, \ldots\}$. We claim that

$$S = S_H \cup \bigcup_P S_P$$

is a $((n_3/4) + O((n_3 + n_2)/\sqrt{r}), \Gamma_r)$-separator of G.

It is clear that S contains at most $n_3/4 + O(n_3/\sqrt{r})$ degree-3 vertices and at most $O(n_2/\sqrt{r})$ degree-2 vertices. Let C be a component of $G \setminus S$. Let d be the number of vertices in C with degree 3. So $d \leq \sqrt{r}$. Since C is connected there are at most $2d + 1$ maximal paths in C that are composed of vertices that have degree-2 in G. Each of these paths can contain at most $(1/5)\sqrt{r}$ vertices. Thus C contains at most

$$(2\sqrt{r} + 1)\frac{1}{5}\sqrt{r} + \sqrt{r} = \frac{2}{5}r + \frac{6}{5}\sqrt{r}$$

vertices. So provided that $r \geq 4$, each component of $G \setminus S$ has at most r vertices. $\square$

2.5.4 Separation lower bound

Define $\mathrm{sep}_G(r)$ by

$$\mathrm{sep}_G(r) = \frac{\min |S|}{|G|}$$

where the minimum is taken over all the $(1, \Gamma_r)$-separators S of G. Let

$$\operatorname{sep}(r) = \max_G \operatorname{sep}_G(r)$$

where G ranges over the set of 3-semiregular graphs. So given any 3-semiregular graph G we can separate it into components of size at most r by deleting at most $\operatorname{sep}(r)|G|$ vertices from G.

Because $\operatorname{sep}(r)$ is decreasing as a function of r, and $\forall r \operatorname{sep}(r) \geq 0$, $\lim_{r\to\infty} \operatorname{sep}(r)$ exists.

From theorem 25 we can immediately deduce:

Corollary 27

$$\lim_{r\to\infty} sep(r) \leq \frac{1}{4}.$$

We will now prove that $\lim_{r\to\infty} \operatorname{sep}(r) = \frac{1}{4}$. This proves that the constant term in Theorem 25 cannot be improved.

Let r and n be any positive integers, $n \geq r$. Margulis [43] constructs a graph $G_{n,r}$ which has the following properties:

- $G_{n,r}$ is 3-regular.
- the order of $G_{n,r}$ is at least n.
- the girth (i.e. length of the shortest cycle) of $G_{n,r}$ is at least $r + 2$.

Lemma 28

$$sep_{G_{n,r}}(r) > \frac{1}{4}$$

Proof: Let S be any $(1, \Gamma_r)$-separator of $G_{n,r}$, and let $C_1, C_2, \ldots, C_l$ be the components of $G_{n,r} \setminus S$.

Consider a component C_i. Let S_i be the set of vertices in S that are adjacent to some vertex in C_i. The restriction on the girth of $G_{n,r}$ implies that C_i is a tree and that each vertex in S_i is adjacent to exactly one vertex in C_i. Since $G_{n,r}$ is 3-regular we have $|S_i| = |C_i| + 2$. Hence

$$\sum_{i=1}^{l} |S_i| = |G_{n,r} \setminus S| + 2l.$$

A vertex in S can be adjacent to at most 3 different components. So it can appear in at most 3 of the S_i sets. Hence

$$\sum_{i=1}^{l} |S_i| \leq 3|S|.$$

So we have that $3|S| \geq |G_{n,r} \setminus S| + 2l > |G_{n,r} \setminus S|$, which shows that more that a quarter of the vertices of $G_{n,r}$ must have been in the separator S. □

Corollary 29

$$\lim_{r \to \infty} sep(r) = \frac{1}{4}.$$

Proof: By Lemma 28, we have that $\mathrm{sep}(r) > \frac{1}{4}$ for all r. Since $\lim \mathrm{sep}(r) \leq \frac{1}{4}$, we have that $\lim \mathrm{sep}(r) = \frac{1}{4}$. □

Chapter 3

Dynamic Fault Diagnosis

In this chapter we consider the *dynamic* model of fault diagnosis that we define in Section 3.1. The difference between the dynamic model and the static model is that the dynamic model allows processors to change status during the diagnosis.

It is impossible to completely diagnose the system because processors continue to break down indefinitely, so any diagnosis cannot hope to reflect the most recent changes. Also it is impossible to guarantee to identify every faulty processor within r rounds of the round in which it first failed. If every processor that tests a fails just before it performs the test, then no reliable processor tests a, and so a remains undiagnosed indefinitely. Instead, the objective for this model is to give an algorithm which can attain a dynamic equilibrium in which processors are being repaired at the same rate that they are failing.

Such an algorithm is given in Section 3.2. The algorithm given there ensures that there are never more than $O(t \log_2 t)$ faulty processors in the system, if up to t processors fail in each round. In Section 3.3 we prove this is the best possible. Some of the material in this chapter originally appeared in [36,37].

3.1 Definitions

In this section we formalize the *ongoing dynamic parallel model* of fault diagnosis. It is similar to the definition of the static model given in section 2.1.

- The computing system is partitioned into n complex indivisible units called *processors* or *nodes*.
- Every processor has a *status*. It can either be *good* or *faulty*. At the start of the procedure all of the processors have *good* status. The status may change if the processor is corrupted or repaired.
- A *test* is an interaction between two processors, a *tester* and a *testee*. The tester claims to determine the status of the testee. If the tester is good then it will correctly diagnose the testee. But if the tester is itself faulty then it has the option of incorrectly diagnosing the testee.
- The operations on the system are performed as a sequence of *rounds*. Each round is divided into three parts. In the first part testing is performed, in the second part some processors are repaired, in the third part some processors are corrupted.
- In the first part of the round every processor may participate in a test, either as tester or testee. But no processor may participate in more than one test at a time. So a round of tests is a directed matching on the processors, but not necessarily a perfect matching.
- In the second part of the round up to t processors may be repaired. When a processor is *repaired*, its status immediately becomes good.
- In the third part of the round, up to t processors may be corrupted. When a processor is *corrupted* its status is immediately changed to faulty. It is possible for a processor to be repaired and then corrupted again in the same round.

The three-part rounds described above will be performed indefinitely. As with the static model, we will view the diagnosis as a contest between a controller and an adversary. In this case the controller's objective is to ensure that there is an upper bound K on the number of faulty processors present at the start of every round, no matter what strategy the adversary follows. The adversary seeks to ensure that there is some round in which more than K processors are faulty.

The phrase "processor a is good in round s" should be taken to mean that a was good at the *start* of round s. It does not mean that a was not corrupted during round s.

3.2 Upper Bound

The object of this section is to describe and analyze an algorithm that the controller can use to perform ongoing diagnosis. We will show that with this algorithm the controller can limit the number of faulty processors present simultaneously in the system to $O(t \log_2 t)$ regardless of the adversary's behavior and the size of the system.

Our algorithm is described in terms of four procedures and the main routine:

DOUBLE combines two sets of processors into a single set of processors by performing tests between the processors in one set and the processors in the other set. It does this in such a way that large sets which consist entirely of good processors can be identified.

GROW starts with singleton sets, and by repeatedly calling DOUBLE it merges the sets in pairs. The procedure takes an argument that tells it how many calls to make to DOUBLE.

SIFT partitions the processors into two parts. One part contains most of the faulty processors, but it may contain good processors as well. The other part consists mostly of good processors; the only faulty processors it contains were corrupted during the call to SIFT. SIFT makes one call to GROW. It uses the sets returned by GROW to decide how to partition the processors.

WINNOW identifies most of the faulty processors. It is given a set of processors that are suspected to be good, and a set of processors that are suspected to be faulty. It tests the faulty suspects with the good suspects. It then calls SIFT, which validates some of the tests by proving that the the tester was good at the moment it performed the test. In this way WINNOW identifies most of the faulty processors in the system. It may overlook some faulty processors, but it never mistakenly identifies a processor as faulty. Once identified as faulty the processor is repaired at the first opportunity. WINNOW returns the set of faulty processors found, along with a new partition of the remaining processors into good suspects and faulty suspects.

ONGOING-DIAGNOSIS consists of an infinite loop that keeps calling WINNOW. Each time it uses the lists of suspects returned by one call to WINNOW to prepare the new lists for the next call to WINNOW. It is the main routine of the algorithm.

In the following subsections we will consider each procedure in turn. Then we will consider the algorithm as a whole and prove that the adversary cannot defeat it.

3.2.1 Double

First we need to define the concept of a helpful set:

Definition 30 Let s_1 and s_2 be two times, $s_2 \geq s_1$. We say that a set of processors A is s_1/s_2*-helpful* if either every processor in A was good at time s_1 or every processor in A is faulty at time s_2. We say that a s_1/s_2-helpful set is helpful at time s_2 with respect to time s_1.

The objective of DOUBLE (Algorithm 3.1) is to combine helpful sets into larger helpful sets. As input, DOUBLE is given sets $C_1, \ldots, C_{2r}$. It pairs them up in some fashion; corresponding to the jth pair it returns a set C'_j which is the union of that pair. Each set C_j is partitioned on input into two parts: A_j, the helpful part, and B_j, *Input:* $(C_1, \ldots, C_{2r}, A_1, \ldots, A_{2r}, B_1, \ldots, B_{2r}, l)$: The C_j are equal-size disjoint sets. Each C_j is partitioned into two sets: A_j and B_j. l is an integer.

Output: $(C'_1, \ldots, C'_r, A'_1, \ldots, A'_r, B'_1, \ldots, B'_r, l')$

Method: Let σ be a permutation of the indices such that

$$0 \leq |A_{\sigma(1)}| \leq |A_{\sigma(2)}| \leq \ldots \leq |A_{\sigma(2r)}| \leq |C_1|. \tag{3.1}$$

Set $C'_j = C_{\sigma(2j-1)} \cup C_{\sigma(2j)}$, for $1 \leq j \leq r$. The procedure takes two rounds. In one round let every processor in $A_{\sigma(2j-1)}$ test a different processor in $A_{\sigma(2j)}$. In the second round, let each processor in $A_{\sigma(2j)}$ that was tested in the first round test the processor from $A_{\sigma(2j-1)}$ that tested it.

Let A'_j consist of those pairs of processors from $A_{\sigma(2j-1)}$ and $A_{\sigma(2j)}$ which liked each other. Let B'_j contain all other processors from C'_j. Set l' to l plus the number of pairs tested where some processor in the pair didn't like the other processor.

Algorithm 3.1: The procedure DOUBLE

the unhelpful part. By performing tests between the sets A_j, DOUBLE can identify a helpful subset A'_j of each set C'_j.

Lemma 31 *Let $C_1, \ldots, C_{2r}, A_1, \ldots, A_{2r}, B_1, \ldots, B_{2r}, l$ and σ have the meaning assigned to them by* DOUBLE. *Let s_0 be a time, and let s be the time of the first round of* DOUBLE. *Then*

(a) If all the sets A_i are s_0/s-helpful, then the sets A'_i are $s_0/(s+2)$-helpful.

(b) If l is a lower bound on the number of faulty processors in $\bigcup B_j$ then l' is a lower bound on the number of faulty processors in $\bigcup B'_j$

Proof: (a) Suppose that all the sets A_j are s_0/s-helpful where s is the first round of DOUBLE. It suffices to show, for each j, that if A'_j contains a faulty processor at time s then all processors in A'_j are faulty in round $s+2$, at the end of DOUBLE.

Suppose $\exists q \in A'_j$ such that q was faulty at time s_0. $q \in A_{\sigma(2j)} \cup A_{\sigma(2j-1)}$, wlog $q \in A_{\sigma(2j)}$. Since $A_{\sigma(2j)}$ is s_0/s-helpful, and contains a processor that was faulty at time s_0, every processor in $A_{\sigma(2j)}$ is faulty at the start of DOUBLE.

Now consider $p \in A'_j$. Either $p \in A_{\sigma(2j)}$ or $p \in A_{\sigma(2j-1)}$. If $p \in A_{\sigma(2j)}$ then p was faulty at the start of DOUBLE, so p is still faulty at the end of the DOUBLE. If $p \in A_{\sigma(2j-1)}$ then p must have liked a processor from $A_{\sigma(2j)}$ as DOUBLE was being executed. Since p liked a faulty processor, p was faulty itself. Thus all of A'_j is faulty at the end of DOUBLE.
(b) If a pair of processors $p_1 \in A_{\sigma(2j)}$, $p_2 \in A_{\sigma(2j-1)}$ test each other and one of them does not like the other, then at least one of them is faulty. l' is l plus the number of such pairs. $\square$

3.2.2 Grow

The purpose of GROW (Algorithm 3.2) is to construct large helpful sets by repeatedly calling DOUBLE. As input, it is given a set M of m processors and an integer d that tells it how many times to call DOUBLE. Initially it views M as m sets, each containing one processor. It returns $m/2^d$ helpful sets, along with the set B of processors that were found to be unhelpful by DOUBLE. We say that the processors in B were *rejected.*

Lemma 32 *Let M, m, d, A^i_j, B^i_j, C^i_j, B and l have the meaning assigned to them by* GROW. *Assume that $2^d \mid m$. Let s_0 be the first round of* GROW. *Then if at most k processors were faulty at the start of* GROW *we have*

(a) The sets A^d_j are $s_0/(s_0+2d)$-helpful.

(b) At least l of the rejected processors are faulty at the end of GROW.

(c) $0 \le l \le k + (2d-1)t$.

(d) $|B| \le 2^d + 2l - 2$.

Proof: (a) By induction on i. Our inductive hypothesis is that for all j, A^i_j is $s_0/(s_0+2i)$-helpful. Since A^0_j contains one element it must be a helpful set. If we have the inductive hypothesis for some i, then by Lemma 31(a) all the A^{i+1}_j returned by the next call to DOUBLE are also helpful with respect to the start of GROW. So the induction goes through and taking $i = d$ proves part (a).

Input: (M,d): M is a set of m processors, d is a positive integer.

Output: $(A_1^d,\ldots,A_{m/2^d}^d,B,l)$

Method: Suppose $M = \{a_1,a_2,\ldots,a_m\}$ and that $2^d \,|\, m$. Define $A_*^i = A_1^i,\ldots,A_{m/2^i}^i$, $B_*^i = B_1^i,\ldots,B_{m/2^i}^i$, and $C_*^i = C_1^i,\ldots,C_{m/2^i}^i$. Set $l_0 = 0$ and

$$\begin{array}{llllll} A_1^0 &= \{a_1\} & A_2^0 &= \{a_2\} & \ldots & A_m^0 = \{a_m\} \\ B_1^0 &= \emptyset & B_2^0 &= \emptyset & \ldots & B_m^0 = \emptyset \\ C_1^0 &= \{a_1\} & C_2^0 &= \{a_2\} & \ldots & C_m^0 = \{a_m\} \end{array}$$

FOR $i = 0$ TO $d-1$
    $(C_*^{i+1},A_*^{i+1},B_*^{i+1},l_{i+1}) =$ DOUBLE(C_*^i,A_*^i,B_*^i,l_i).
SET $B = \bigcup_j B_j^d$.
SET $l = l_d$.

Algorithm 3.2: The procedure GROW

(b) Our inductive hypothesis is that l_i is a lower bound on the number of faulty processors in $\bigcup B_j^i$. The base case is trivial since $l_0 = 0$ and $\bigcup B_j^0 = \emptyset$. If we have the inductive hypothesis for some i then Lemma 31(b) shows it is true for $i+1$.

(c) By assumption there are at most k faulty processors in M at the start of GROW. Since GROW takes $2d$ rounds to perform, M contains at most $k+2dt$ faulty processors at the end of GROW. Because tests are performed in the first part of a round, and corruptions are performed in the third part of a round, it is impossible that corruptions that occurred in the last round of GROW were detected in any way. So we have that $l \leq k + (2d-1)t$.

(d) There are two ways a node could be rejected. Either it is in a test in which one node does not like the other one, or for some i, j it is one of the $|A_{\sigma_i(2j)}^i| - |A_{\sigma_i(2j-1)}^i|$ nodes from $A_{\sigma_i(2j)}^i$ which are rejected because there are no nodes in $A_{\sigma_i(2j-1)}^i$ to pair them with. There are exactly $2l$ nodes which are rejected for the first reason. The choice of σ_i described by equation (3.1) ensures that on the ith call to DOUBLE at most $|C^{i-1}| = 2^{i-1}$ nodes can be rejected for the second reason. However when $i = 1$, *Input:* (M, d, k): M is a set of m processors, d is an integer, and k is an upper bound on the number of faulty processors in M.

Output: (G, F): A partition of M.

Method:

SET $(A_1^d, \ldots, A_{m/2^d}^d, B, l) = \text{GROW}(M, d)$.
SET $G = \bigcup_j \{A_j^d : |A_j^d| > k + (2d-1)t - l\}$.
SET $F = M \setminus G$.

Algorithm 3.3: The procedure SIFT

every A_j^1 is a single processor, so no processors are rejected on the first call to DOUBLE. Summing, we see that at most $2l + 2 + 4 + \cdots + 2^{d-1} = 2^d + 2l - 2$ nodes can be rejected. □

3.2.3 Sift

The purpose of SIFT (Algorithm 3.3) is to partition the set M of processors into two parts: G and F. All the processors in G were good as the start of SIFT. All of the processors that were faulty at the start of SIFT are in F. G may contain some faulty processors, since some processors will be corrupted during the execution of SIFT. F may contain some good processors but we will bound the number of good processors in F.

SIFT works by calling GROW. It tells GROW to call DOUBLE sufficiently often that some of the helpful sets A_j^d returned by GROW are larger than the number of faulty nodes in the system. Since these helpful sets must contain a good member at the end of SIFT, all of their members were good at the start of SIFT. G is set to the union of these sets, and F is $M \setminus G$. We say that G is the set of processors *accepted* by SIFT and F is the set of processors *rejected* by SIFT. We say that A_j^d is *rejected* if its members are placed in F.

We need to prove a bound on the number of good processors that get placed in F.

This bound will depend on the number of processors t corrupted each round, the number of times d that DOUBLE is called, and an upper bound k on the number of faulty processors in M at the start of SIFT. Make the following definition:

$$E(d,k,t) = \begin{cases} 2^d + 2k + 2(2d-1)t - 2 & \text{if} \quad 2k + 2(2d-1)t - 2 < 2^d \\ \dfrac{(2^d-2)2^d}{2^d - k - (2d-1)t} & \text{if} \quad k + (2d-1)t < 2^d \\ & \text{and} \quad 2k + 2(2d-1)t - 2 \geq 2^d \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.2}$$

Lemma 33 *Let M, m, d, k, G, F, A_j^d, and B have the meaning assigned to them by* SIFT. *Assume that $2^d \mid m$, and that at most k processors were faulty at the start of* SIFT. *Let s_0 be the first round of* GROW. *Then we have*

(a) If $p \in G$ then p was good at the start of SIFT.

(b) $|F| \leq E(d,k,t)$.

Proof: (a) Suppose $p \in M$ and p was faulty at the start of SIFT. Either $p \in B$ or for some j, $p \in A_j^d$. In the former case we have immediately that $p \in F$ since $B \subseteq F$. In the latter case, since by Lemma 32(a) A_j^d is a $s_0/(s_0+2d)$-helpful set, all of the processors in A_j^d are faulty at the end of SIFT. There are at most $k+(2d-1)t$ faulty processors in M, and by Lemma 32(b) at least l of them are in B which is disjoint from A_j^d. So

$$|A_j^d| \leq k + (2d-1)t - l. \tag{3.3}$$

SIFT places all A_j^d that satisfy (3.3) in F.

(b) Suppose A_j^d is rejected. Then since $|A_j^d| + |B_j^d| = 2^d$ we have

$$|B_j^d| \geq 2^d - k - (2d-1)t + l. \tag{3.4}$$

Let r be the number of A_j^d that are rejected. Then by (3.4)

$$r(2^d - k - (2d-1)t + l) \quad \leq \sum_{j:A_j^d \text{ rejected}} |B_j^d|$$

$$\begin{aligned} &\leq |B| \\ &\leq 2^d + 2l - 2 \end{aligned}$$

where the last inequality is Lemma 32(d).

Hence

$$r \leq \frac{2^d + 2l - 2}{2^d - k - (2d-1)t + l} \tag{3.5}$$

Using (Lemma 32(d), 3.5, 3.3) we obtain the following bound on $|F|$:

$$\begin{aligned} |F| &= |B| + \sum_{j : A_j^d \text{ rejected}} |A_j^d| \\ &\leq 2^d + 2l - 2 + \frac{2^d + 2l - 2}{2^d - k - (2d-1)t + l} \cdot (k + (2d-1)t - l) \\ &= \frac{(2^d + 2l - 2)2^d}{2^d - k - (2d-1)t + l}. \end{aligned} \tag{3.6}$$

Since the adversary can control the size of l we wish to remove the dependence on l from this bound. Let $f(l)$ be the expression on the right hand side of (3.6). Lemma 32(c) states that $0 \leq l \leq k + (2d-1)t$. It is easy to see that $f(l)$ is monotone within this range. If $2k + 2(2d-1)t - 2 \leq 2^d$ then f is increasing and so it is maximized by taking $l = k + (2d-1)t$. In the other case f is decreasing, and so $f(l)$ is maximized by setting $l = 0$. Substituting these values for l in (3.6) gives $E(d, k, t)$. Thus we have proved part (b). □

3.2.4 Winnow

The purpose of WINNOW is to discover faulty processors. To prove that a processor is faulty it suffices to test it, and then use SIFT to verify that the tester was good when the test was performed.

This suggests the WINNOW procedure (Algorithm 3.4). WINNOW takes as its argument three sets of processors:

Input: (T, U, Z, d, k_T): T, U, Z are disjoint sets of processors satisfying $|U| \leq |T|$ and $|Z| \leq (2d+1)t$. Z is a set of faulty processors. d is an integer such that $2^d \,\big|\, |T|$. k_T is a bound on the number of faulty processors in T at the start of WINNOW.

Output: (T', U', Z')

Method: Pair each processor in U with some processor in T. In one round, have every processor in T test its partner in U if it has one.

Let $(G, F) = \textsc{sift}(T, d, k_T + t)$.

Spend the repair parts of these $2d+1$ rounds repairing every processor in Z.

Let U_u consist of those processors in U which were paired with some processor in F. Partition $U \setminus U_u$ into U_f and U_g where U_g contains those processors in U whose partner in G diagnosed that they were good, and U_f contains those whose partner diagnosed that they were faulty.

Set $T' = G \cup Z \cup U_g$, $U' = F \cup U_u$ and $Z' = U_f$.

Algorithm 3.4: The procedure WINNOW

Symbol	**Meaning**
T	mostly good processors to perform the diagnosis
U	processors whose status is to be determined
Z	faulty processors to be repaired

The parameter d determines how many rounds of DOUBLE take place. k_T bounds the number of faulty processors present in T at the start of WINNOW.

The result of WINNOW is another partition of $T \cup U \cup Z$. This partition classifies the processors depending on what WINNOW has diagnosed about their status. The sets returned satisfy the properties summarized in the following table:

Symbol	**Property**
T'	were all good at some point during the application of WINNOW
U'	have unknown status
Z'	are all faulty at the end of WINNOW

Let $\nu(A)$ denote the number of faulty processors in a set A at the *start* of WINNOW. Let $\nu'(A)$ denote the number of faulty processors in A at the *end* of WINNOW.

Lemma 34 *The sets returned by* WINNOW(T, U, Z, d, k_T) *satisfy*

(a) $\nu'(T') \leq (2d+1)t$.

(b) $|U'| \leq 2E(d, k_T + t, t)$.

(c) $\nu(U) - E(d, k_T + t, t) \leq |Z'| \leq \nu(U)$.

(d) $\nu'(T' \cup U') \leq \nu(T \cup U) + 2dt + t$.

Proof: (a) Every processor in T' either was good at the beginning of the execution of SIFT (those processors from G and U_g) or was repaired while WINNOW was executing. So if a processor in T' is faulty at the end of the procedure it must have changed status to faulty during the procedure. Since WINNOW takes $2d+1$ rounds to perform, at most $(2d+1)t$ processors could have been corrupted.

(b) Observe that $U' = F \cup U_u$. But $|U_u| = |F|$, so $|U'| = 2|F|$. Since WINNOW spends its first round having T test U, there are at most $k_T + t$ faulty processors in T when SIFT is called. The bound on $|F|$ is given by Lemma 33(b).

(c) Since WINNOW does not discover processors that were corrupted during its execution, $|Z'| \leq \nu(U)$. The only way that WINNOW fails to discover a faulty processor is if its tester was placed in F by SIFT. Thus $\nu(U) - |Z'| \leq |F|$. By Lemma 33(b) $|F| \leq E(d, k_T + t, t)$.

(d) Suppose that $p \in T' \cup U'$ and that p is faulty at the end of WINNOW. Then either p was corrupted during the execution of WINNOW (this happens to at most $2dt + t$ processors) or p was faulty at the start of WINNOW and $p \in T \cup U$ (this is true for exactly $\nu(T \cup U)$ processors). □

3.2.5 Ongoing diagnosis algorithm

ONGOING-DIAGNOSIS (Algorithm 3.5) repeatedly calls WINNOW. Given the sets T', U' and Z' returned by WINNOW, it constructs new sets T'', U'' and Z'' suitable for the next call to WINNOW. The set T will have a fixed size, s_T, which is the same for every call to WINNOW. We will see shortly how to choose s_T for a given value of n.

Input: (N, t, s_T, d, k_T): N is a set of n good processors. t is the maximum number of processors that may be corrupted or repaired each round. s_T, d, and k_T are integers.

Method:

SET $T'' \subseteq N$, $|T''| = s_T$, $U'' = N \setminus T''$, $Z'' = \emptyset$.
infinite LOOP
  SET $T = T''$, $U = U''$, and $Z = Z''$.
  SET $(T', U', Z') =$ WINNOW(T, U, Z, d, k_T).
  SET $T'' \subseteq T'$, $|T''| = s_T$.
  SET $Z'' \subseteq Z'$, $|Z''| = \min\{|Z'|, (2d+1)t\}$.
  SET $U'' = U' \cup (T' \setminus T'') \cup (Z' \setminus Z'')$.

Algorithm 3.5: ONGOING-DIAGNOSIS

Z will always contain processors that are known to be faulty, but never more than the $(2d+1)t$ processors that can be repaired during one call to WINNOW. U will contain the remaining processors.

For the algorithm to work correctly it suffices to choose s_T, d, and k_T so that $|T'| \geq s_T$. We also need to prove a bound on the number of faulty processors in the system which depends only on t and not upon n.

Lemma 35 *Let T and d have the meaning assigned to them by* ONGOING-DIAGNOSIS*. Then $\nu(T) \leq 2dt + t$ at all times.*

Proof: Lemma 34(a) states that $\nu'(T') \leq 2dt + t$. Since $T'' \subseteq T'$ we have $\nu'(T'') \leq 2dt + t$ also. On the very first call to WINNOW, $\nu(T) = 0$ because at the start of ONGOING-DIAGNOSIS all the processors are good. In all subsequent calls to WINNOW T is set to the T'' constructed from the previous call. So we have the stated bound. □

Recall that k_T is a bound on the number of faulty processors in T on a call to WINNOW. The parameter k to GROW is an upper bound on the number of faulty processors in M at the start of a call to GROW. WINNOW sets $k = k_T + t$. In view of Lemma 35 we will define:

$$\begin{aligned} k_T &= 2dt + t \\ k &= 2dt + 2t \end{aligned} \tag{3.7}$$

The following lemma will provide an upper bound on the number of faulty processors in $T \cup U$ at the start of each call to WINNOW made by ONGOING-DIAGNOSIS.

Lemma 36 *Suppose before a call of* WINNOW *we have*

$$\nu(T \cup U) < E(d, 2dt + 2t, t) + 6dt + 3t.$$

Then after the call we have

$$\nu'(T'' \cup U'') < E(d, 2dt + 2t, t) + 6dt + 3t.$$

Proof: We will divide the proof into two cases.

1. $|Z'| < 2dt + t$: That is, not enough faulty processors were diagnosed to be able to repair the full amount permitted during the next call to WINNOW.

   Then by lemma 34(c)

   $$\nu(U) \leq E(d, k_T + t, t) + |Z'| < E(d, 2dt + 2t, t) + 2dt + t.$$

   Thus, by Lemma 35 we obtain

   $$\nu(T \cup U) < E(d, 2dt + 2t, t) + 4dt + 2t.$$

   By Lemma 34(d) we obtain

   $$\nu'(T' \cup U') < E(d, 2dt + 2t, t) + 6dt + 3t.$$

   When $|Z'| < 2dt + t$ we have $T'' \cup U'' \subseteq T' \cup U'$, so

   $$\nu'(T'' \cup U'') \leq \nu'(T' \cup U') < E(d, 2dt + 2t, t) + 6dt + 3t.$$

2. $|Z'| \geq 2dt + t$: Therefore $|Z''| = 2dt + t$. Then we have discovered and removed at least $2dt + t$ faulty processors from $T \cup U$ to construct $T'' \cup U''$. Since this is the maximum number of processors that get corrupted during a call to WINNOW we have
$$\nu'(T'' \cup U'') \leq \nu(T \cup U).$$ □

Next we select a suitable value for d. The best value is one that minimizes $E(d, 2dt + 2t, t)$. A good value of d (although not always the best for specific values of t) is obtained by taking d to be the smallest integer such that

$$2^{d-1} \leq 2k + 2(2d-1)t - 2 = 8dt + 2t - 2 \leq 2^d. \tag{3.8}$$

We can estimate d in terms of t. As $t \to \infty$ we can see that $d \to \log_2 t$ from above. For example provided $t > 70$, we have

$$\begin{aligned} d &\leq 2\log_2 t, \\ 2^d &\leq 16dt + 4t - 4 \leq 32t\log_2 t + 4t - 4 \end{aligned} \tag{3.9}$$

where the latter inequality follows from (3.8).

The value we have chosen for d ensures that

$$E(d, 2dt + 2t, t) = 2^d + 8dt + 2t - 2 \tag{3.10}$$

By substituting (3.10) into lemmas 34 and 36 we get

$$\begin{aligned} |U'| &\leq 2 \cdot 2^d + 16dt + 4t - 4, \\ |Z'| &\leq \nu(U) < 2^d + 14dt + 5t - 2, \\ |U'| + |Z'| &< 3 \cdot 2^d + 30dt + 9t - 6, \\ |T'| &= n - (|U'| + |Z'|) \\ &> n - (3 \cdot 2^d + 30dt + 9t - 6) \end{aligned} \tag{3.11}$$

Theorem 37 *Let $t > 70$ and $n \geq 376t\log_2 t + 50t$. Then we can perform ongoing diagnosis so that at most $64t\log_2 t + 10t$ processors are ever simultaneously faulty.*

Proof: ONGOING-DIAGNOSIS requires that $|T| \leq |T'|$ and $|U| \leq |T|$. SIFT requires that $2^d \,\big|\, |T|$. We define s_T to be the largest multiple of 2^d which is smaller than $n - (3 \cdot 2^d + 30dt + 9t - 6)$.

We still need to show that $|U| \leq s_T$. We have $|T| + |U| + |Z| = n$. Since it is possible that $|Z| = 0$, it will suffice to show that $s_T \geq n/2$.

Since $n \geq 376t \log_2 t + 50t$ we have that

$$n \geq 8 \cdot 2^d + 60dt + 18t - 12 \tag{3.12}$$

Rearranging (3.12) we obtain

$$\frac{n}{2} \leq n - (3 \cdot 2^d + 30dt + 9t - 6) - 2^d.$$

Thus $s_T \geq n/2$ as required.

On the first call to WINNOW $\nu(T \cup U \cup Z) = 0$. Lemma 36 gives the maximum number of faulty processors present in $T'' \cup U''$ at the end of each call to WINNOW. Since Z'' can contain at most $2dt + t$ faulty processors we have that on all calls to WINNOW after the first,

$$\nu(T \cup U \cup Z) \leq E(d, 2dt + 2t, t) + 8dt + 4t \leq 2^d + 16dt + 6t - 2$$

By assuming that when $|Z| < 2dt+t$, WINNOW performs its repairs as soon as possible, we see that this bound must apply at all times. Applying (3.9) gives the required result. □

Although Theorem 37 assumes that $t > 70$, the ongoing diagnosis algorithm will work for any t. The only place where we used the assumption that $t > 70$ was in the derivation (3.9) of the upper bounds on d as a function of t. It is easy to find a suitable value of d for any particular t. The following example considers the case $t = 1$.

Example 38 *Let $t = 1$. Take $d = 5$, so $E(d, 2dt + 2t, t) = 87$. Using the same method of derivation as (3.11) we obtain*

$$|T| \leq n - (174 + 120) = n - 294.$$

The smallest n such that $|T| \geq n/2$ and $32 \,\big|\, |T|$ is given by

$$n = 640 \quad and \quad |T| = 320.$$

Thus we have an algorithm for ongoing diagnosis with $n = 640$ and $t = 1$ which ensures that there will never be more than 120 faulty processors.

3.3 Lower Bound for Ongoing Diagnosis

The aim of this section is to prove that in the ongoing diagnosis model it is impossible for the controller to prevent $t \log_2 t$ processors from being simultaneously faulty. This proves that Theorem 37 is optimal up to a constant factor. Obviously we cannot prove that $t \log_2 t$ processors become faulty if $n < t \log_2 t$. We shall assume that $n \geq 4t \log_2 t$.

We will view the ongoing diagnosis as a series of *phases*. From time to time the adversary will declare the start of a new phase by revealing to the controller the current status of every processor in the system. Thus the controller starts each phase knowing the entire state of the system. Nevertheless we will show that although the adversary corrupts t processors in each round of the phase, one of the following statements must be true during the phase:

1. The controller repairs a processor which in fact had good status.

2. There is a round in which the controller repairs fewer than t processors.

3. At least $t \log_2 t$ processors were faulty at the start of the phase.

It is clear that each phase in which statement 1 or 2 is true increases the number of faulty processors in the system by one. So after at most $t \log_2 t$ phases, statement 3 will be true.

The adversary has immediately achieved its aim if the controller fails to repair t processors in some round. Thus in the following sections we will assume that the controller repairs t processors in every round.

At most t processors are repaired and corrupted each round.

FOR each round of the phase:
START round i.
test part: IF (faulty v tests any u) v REPORTS u is faulty.
repair part: SET R_i to the set of processors the controller repairs.
corrupt part: IF ($|R_i| \geq t - 2^{i-1}$ and $i > 1$)
SET S_i to the first $\max\{t - 2^{i-1}, 0\}$ elements of R_i.
ELSE SET $S_i = R_i$.
CORRUPT S_i.
END round i.

Algorithm 3.6: The adversary's designated strategy

3.3.1 The adversary's strategy

As a phase proceeds, there are two ways in which the adversary affects the state of the controller and the state of the system. First it chooses which processors to corrupt in each round. Second it chooses which state a faulty processor reports when it performs a test.

We will consider two strategies that the adversary could use to make these choices: the *designated* strategy and the *true* strategy. These strategies will be *consistent* in the sense that although in some round a processor could have one status under the designated strategy and the other status under the true strategy, the controller will see precisely the same test results regardless of which strategy the adversary is following. Thus the controller will be unable to distinguish between the two strategies.

Definition 39 A processor p is an *original fault* in round i of a phase if p was faulty at the start of the phase and it was not repaired in the rounds before round i.

The adversary will actually follow the true strategy, but we will first consider the designated strategy which is defined by Algorithm 3.6. Let k be the first round in which the controller would repair a processor that is not an original fault in that round, supposing that the adversary followed the designated strategy. Let ρ be a processor that the controller would repair in round k, that is not an original fault in round k. It is possible that ρ was faulty at the start of the phase, and that the controller would repair ρ for the *second* time in round k. A processor's status in a given round when the adversary follows the designated strategy is called the processor's *designated status.* We will represent a designated status by placing quotation marks around it.

Since the controller is deterministic and the adversary is omniscient, the adversary can precisely predict what the controller will do if the adversary were to follow the designated strategy. In particular the adversary can calculate k, ρ, and all of the R_i and S_i as a function of the set of faulty processors at the start of the phase.

Given this knowledge, the adversary can construct a true strategy which is consistent with the designated strategy. During the first $\lceil \log_2 t \rceil$ rounds, the true strategy corrupts exactly t processors per round but does not corrupt ρ. The adversary announces the start of the next phase at the *end* of round k.

Suppose $k < \log_2 t$. Then, since ρ cannot be faulty in round k, the controller repairs a good processor during the phase. But if $k \geq \log_2 t$ and the controller repairs t processors every round, then the controller repaired $t \log_2 t$ processors during the phase, all of which were faulty at the start of the phase. So one of the three statements listed in Section 3.3 must be true during the phase.

3.3.2 The adversary's true strategy

Now we determine a true strategy with the properties listed in the last section. We will refer to a processor's status under the true strategy as its *true status.* We will call a processor a *masquerade* in a given round if its true status and its designated status are different.

Let M_i denote the set of masquerades in round i. Since the adversary wants to follow a consistent strategy, it wants to have as few masquerades as possible. It will greatly simplify matters if the adversary follows a strategy in which the only masquerade which is designated "faulty" (and thus is truly good) is ρ. Recall that a processor's status in a round is its status at the beginning of the round.

Lemma 40 *Assume that ρ is the only masquerade designated "faulty".*

(a) Let $1 \leq i \leq k-1$. Then $M_i \subseteq M_{i+1}$.

(b) $M_1 = \emptyset$. Let $1 < i \leq k$. If $\rho \notin M_i$ then $|M_i| = 2^{i-1} - 2$. If $\rho \in M_i$ then $|M_i| = 2^{i-1}$.

Proof: (a) Let $p \in M_i$ and consider two cases: First suppose p has designated status "faulty" in round i, so $p = \rho$. By the definition of ρ the controller will not repair p until round k, so p will have designated status "faulty" in round $i+1$. Since a true strategy never corrupts ρ, p's true status in round $i+1$ will be good. So $p \in M_{i+1}$.

Otherwise p has designated status "good" in round i. But in the designated strategy no processor ever changes state from "good" to "faulty", since $S_i \subseteq R_i$. So p will have designated status "good" in round $i+1$. In rounds 1 to $k-1$ the controller repairs only original faults which must have designated status "faulty". p has designated status "good" so it will not be repaired. Thus its true status will be faulty in round $i+1$, and so $p \in M_{i+1}$.

(b) Consider the number of faulty processors in round i under each strategy. Let x_i be the number of faulty processors in round i under the designated strategy, and x'_i the number of faulty processors under the true strategy. Then $x_1 = x'_1$.

Under the designated strategy, t processors are corrupted in round 1, and $t - 2^{i-1}$ processors are corrupted in round i for $i > 1$. So for $i > 1$ we have $x_{i+1} = x_i - 2^{i-1}$ which implies $x_i = x_1 - 2^{i-1} + 2$.

Under the true strategy t processors are corrupted every round. So $x'_i = x'_1$.

Figure 3.7 shows that if μ_i is the number of masquerades that are designated faulty in round i, then for $i > 1$ we have

$$|M_i| = 2\mu_i + x'_i - x_i = 2\mu_i + 2^{i-1} - 2.$$

If $\rho \notin M_i$ then $\mu_i = 0$ and $|M_i| = 2^{i-1} - 2$. If $\rho \in M_i$ then $\mu_i = 1$ and $|M_i| = 2^{i-1}$. $\square$

	truly good	**truly faulty**	**total**
designated "good"	$n - x'_i - \mu_i$	$x'_i - x_i + \mu_i$	$n - x_i$
designated "faulty"	μ_i	$x_i - \mu_i$	x_i
total	$n - x'_i$	x'_i	n

The total number of masquerades is $2\mu_i + x'_i - x_i$.

Figure 3.7: The number of masquerades in a round. x_i is the number of processors that have designated status "faulty", x'_i is the number of processors that have true status faulty, and μ_i is the number of masquerades that have designated status "faulty".

We will now describe which processors the adversary corrupts when it follows its true strategy. Since we have already given the designated strategy, the true strategy will be completely determined by describing which processors are masquerades in each round. We will state how the adversary selects M_{i+1}, given that it has already selected M_i. Since the designated and true status of a processor are the same in round 1 of a phase, we have that $M_1 = \emptyset$.

We have already indirectly described some aspects of the adversary's choice for M_{i+1}. The statement that the adversary does not corrupt ρ during the phase, means that ρ will become a masquerade in round $i + 1$, if it is corrupted in round i under the designated strategy. The choice of the designated strategy, and the decision that no processor other than ρ is ever a masquerade that is designated "faulty", means that the masquerades will satisfy the properties of Lemma 40.

Thus we have already indirectly specified some of the members of M_{i+1}. First by Lemma 40(a) we have that $M_i \subseteq M_{i+1}$. Second if the controller repairs ρ in round i then $\rho \in M_{i+1}$. By Lemma 40(b) we see that we have already specified at most 1/2 of the members of M_{i+1}. Thus for each choice of a masquerade in round $i + 1$ that we have already specified, there is another choice of a masquerade which is still open. We will show that these other masquerades can be chosen so that they ensure that the true strategy is consistent. Intuitively, whenever there is danger of a masquerade being discovered by some test, we make the tester into a masquerade (i.e. corrupt it)

Round i test			**Status of testee u**			
			Designated "good"		**Designated "faulty"**	
			truly good	**truly faulty**	**truly good**	**truly faulty**
Status of tester v	**designated "good"**	**truly good**	good $\surd$	good $\times$	faulty $\times$	faulty $\surd$
		truly faulty	good $\surd$	good $\surd$	faulty $\surd$	faulty $\surd$
	designated "faulty"	**truly good**	faulty $\times$	faulty $\surd$	faulty $\times$	faulty $\surd$
		truly faulty	faulty $\surd$	faulty $\surd$	faulty $\surd$	faulty $\surd$

Figure 3.8: Combinations of true and designated status

so that it can give consistent test results.

Figure 3.8 shows every possible combination of the true and designated status of the tester v and the testee u in a test that is performed in round $i+1$. The entry in the table states the result that the controller would see if the adversary was following the designated strategy. An entry is marked with a a $\surd$ if and only if the adversary can give the required test result. Since the adversary seeks a consistent strategy, it must avoid the entries marked with an $\times$.

From Figure 3.8 we see that if the tester v is truly faulty then the test will be consistent. So if $v \neq \rho$ and $u \in M_i$, it is sufficient for the adversary to corrupt v in round i (if v isn't already faulty). Similarly if $v = \rho$ and is designated "faulty" in round $i+1$, then it is sufficient for the adversary to corrupt u in round i, (if u isn't already faulty).

The adversary must ensure that no test occurs in which $v = \rho$ and u is a masquerade (which is necessarily a faulty processor) when ρ is designated "good". In this case the adversary would gain nothing by corrupting the testee (indeed it is already faulty), nor can it corrupt the tester since it is ρ.

It avoids such a test by planning ahead so it doesn't occur. If it ever happens that ρ is a masquerade, then the problem cannot occur in subsequent rounds because ρ will never be designated "good" in the future. As long as ρ doesn't become a masquerade, the only way that the adversary can be forced to make a processor into a masquerade is if it tests a processor that was made into a masquerade in an earlier round. So by examining the testing graphs the adversary can find all the processors which, if they were made into masquerades, would force ρ to test a masquerade in a later round. As long as it avoids corrupting any of these processors, it will obtain a consistent strategy. Define the sets $D_k, D_{k-1}, \ldots, D_1$ recursively:

- $D_k = \{u : \rho \text{ tests } u \text{ in round } k\}$.
- $D_i = D_{i+1} \cup \{v : v \text{ tests } u \in D_{i+1} \text{ or } \rho \text{ tests } v \text{ in round } i\}$.

Provided that in any round in which ρ is not a masquerade we have $M_i \cap D_i = \emptyset$, then the true strategy will be consistent with the designated strategy.

We have shown that for each choice of a masquerade that was previously specified, the adversary also has a choice available that was not previously specified, which it then uses, if it is needed, to obtain a consistent strategy. But Lemma 40 states exactly how many masquerades there will be in each round. So we may not have specified all of the choices of masquerades that the adversary must make.

If there are any choices left over the adversary may choose any processor to corrupt in round i provided that it is not designated "faulty", it is not in D_{i+1} and it is not tested by a truly good processor in round $i+1$. We need to show that there will always be enough suitable processors to choose.

Since we are assuming that statement 3 is not true, fewer than $t \log_2 t$ processors have been designated "faulty". From its definition we see that $|D_{i+1}| \leq 2^{k-1}$. At most $n/2$ processors can be tested in round $i+1$. Since by Lemma 40(b) there are at most $2^{i-1} \leq 2^{k-1}$ masquerades, we have that at most

$$t \log_2 t + 2^{k-1} + \frac{n}{2} + 2^{k-1}$$

processors can be ruled out as suitable choices of masquerades.

Theorem 41 *Let $n \geq 4t \log_2 t$. There is no deterministic dynamic fault diagnosis algorithm that limits the number of faulty processors to $t \log_2 t - 1$ at all times.*

Proof: The theorem is trivially true if $t = 1$. If $t > 1$, then since $k < \log_2 t$ we have that $2^k < t \log_2 t$. Since $n \geq 4t \log_2 t$ we have

$$n \geq 2 \cdot 2^k + 2t \log_2 t.$$

Rearranging this inequality gives

$$n - \left(t \log_2 t + 2^{k-1} + \frac{n}{2} + 2^{k-1} \right) \geq 0.$$

From the discussion above this suffices to proof the theorem. □

3.4 Conclusion

We have shown in Theorem 37 that no matter how large n is, there is an algorithm for the controller in which $O(t \log_2 t)$ processors become simultaneously faulty. Section 3.3 shows that there is no algorithm for the controller that can prevent $t \log_2 t$ processors from becoming simultaneously faulty. So Theorem 37 is optimal, up to a constant factor.

Observe that ONGOING-DIAGNOSIS never repairs a good processor. It really is diagnosing the system. Since the adversary presented in Section 3.3 corrupts t processors every round, it will not help the controller if it is told how many processors are corrupted each round. Similarly it will be no help to the controller if it isn't required to *prove* a processor is faulty before it repairs it. (It might seem that this is useful in situations where the adversary doesn't corrupt t processors in some round.) So any algorithm for ongoing diagnosis must be able to prove that a processor is faulty before it repairs it.

It is not hard to modify the adversary argument in Section 3.3 to apply without the restriction $n \geq 4t \log_2 t$. We find that for smaller n we can prove that eventually at least one ninth of the processors must become simultaneously faulty.

However if is interesting to note that if we consider a changed model in which the controller is permitted to repair $2t$ processors each round, and isn't required to prove that a processor is faulty before repairing it, then an algorithm can be found in which the bound on the total number of faulty processors simultaneously present is linear in t. This algorithm will appear in a future paper.

Chapter 4

Distributed Diagnosis

In this chapter we are interested in fault diagnosis in a distributed setting. Instead of having an external *controller* that directs the diagnosis, the processors will act independently. Each processor will have its own internal table giving the (most recently seen) status of every other processor in the system. From time to time a processor can test the status of another processor. It adds the result of the test to its internal table. It can also use any additional information that it obtains from the processor it tested.

Observe that this problem has quite a different flavor from the static and dynamic fault diagnosis problems. In the previous chapters the difficulty of the problem was deciding the status of the processors. But in this case the tester *knows* the status of the tested processor, since it just tested it. We are only interested in showing that good processors will correctly diagnose the system. If the tester is good, then the result of its test is accurate. If the tester is faulty, then we are not interested in the conclusions that the tester reaches about the testee's status.

Instead the problem becomes one of communication. How can we ensure that the good processors will quickly arrive at a correct diagnosis? How quickly can they update the diagnosis if the status of some processors changes?

In Section 4.1 we will formalize a *distributed* model of fault diagnosis. In Sections 4.2 and 4.3 we will discuss the *rumor spreading* and the *single-shot cooperative-collect* problems. We will present and analyze new algorithms for these problems, which involve the spread of information in an asynchronous distributed environment. We will show how to use them to achieve efficient fault diagnosis. Finally is Section 4.4 we will consider the dynamic version of these algorithms and in Section 4.5 we will note what happens if the processors all operate at the same speed.

4.1 Definitions

In this section we will define several models of distributed fault diagnosis. We will also define some problems from distributed computing which are closely related to them.

All *distributed fault diagnosis* models will possess the following properties:

- The computing system is considered to be partitioned into n complex indivisible units called *processors* or *nodes*.
- At any time each processor has a *status* which will be *good* or *faulty*. The object of the algorithm is for each good processor to determine the status of every other processor in the system.
- There is no upper bound on the number of faulty processors.
- A *test* is an interaction between two processors, a *tester* and a *testee*. The tester determines the status of the testee. We will assume that the testee gains no knowledge at all about the tester. In fact we will make the most general assumption and suppose that the testee is unaware that it has been tested.
- Each processor possesses its own single-writer multi-reader register. Only the owner of a register may write to it, but any processor can read from it. Normally a processor will not trust any information written on another processor's register.

As part of the testing operation a tester will read the testee's register. If the testee was good, the tester is certain that all the information the register contained was accurate, and it may make use of this information. If the testee was faulty, the tester cannot rely upon any of the information in the register. In the latter case, the only knowledge that the tester gains from the testing operation is that the testee is faulty.

Definition 42 In the *static* distributed diagnosis model every processor has a fixed status. In the *dynamic* distributed diagnosis model the status of a processor may change during the diagnosis.

No algorithm can perfectly solve the dynamic distributed diagnosis model, because it is always possible that a processor has changed status since it was last tested. Instead we will give an algorithm which can obtain a *fresh* diagnosis. At any time a processor may start a new diagnosis operation. At the end of the operation the processor will have obtained a status for each other processor in the system, that is fresh in the sense that it was accurate at some point during the diagnosis operation.

4.1.1 Modeling the passage of time

A major feature of distributed models is how they represent the passage of time. We will introduce two variants: an asynchronous model in which we suppose that the processors are operating at different speeds and a synchronous model in which we suppose that the processors are operating at the same speed.

Definition 43 In the *asynchronous* distributed diagnosis model only one processor may be executing at any time. There is an adversary which acts as the scheduler. From time to time the adversary may preempt the processor that is executing, and select a different processor to resume operation. A processor cannot determine when it has been preempted, nor can it determine which other processors have been in operation.

In the *synchronous* distributed diagnosis model the testing proceeds in rounds. In each round every processor may select another processor to test. There is no scheduler in the synchronous model. If several testers simultaneously select the same processor to test then the tests conflict and none of them are performed. If a processor decides to perform a test and is also selected as the testee by another processor then the tests conflict and neither of them is performed. All the tests that are performed in one round are deemed to happen simultaneously.

For each of these two models we must decide how to measure the cost of performing an algorithm. For the asynchronous model, in which some processors may be moving much faster than other processors, we measure the cost of an algorithm by counting the number of tests performed as part of the algorithm by good processors.

Since the actions of faulty processors do not contribute to the algorithm there is no point in counting them. Indeed if we did count faulty processors the adversary could make an algorithm appear to take arbitrarily long to complete by only scheduling faulty processors. Once a good processor has finished the algorithm we assume that when it is scheduled it spends its time performing some other task. So again we should not count this time as part of the cost of performing the algorithm.

It is a reasonable assumption that the testing of one processor by another, and the communication overhead of copying information from testee to tester, require much more processing time than the time that it takes a processor to decide which processor to test next. This justifies only counting the number of tests that a processor performs. Since we will be considering randomized algorithms these numbers will be upper bounds that are stated with high probability.

For the synchronous model, in which all the processors are moving simultaneously, we measure the cost of an algorithm by counting the number of rounds of tests performed before all the good processors have completed the algorithm. Since the faulty processors might never finish the algorithm there is no point in waiting for them to finish.

Note that we are counting individual tests in the asynchronous model and rounds of tests in the synchronous model. Thus we expect the cost of an asynchronous algorithm to be a factor of n greater than the cost of a synchronous algorithm.

4.1.2 The rumor-spreading problem

The rumor-spreading problem is the simplest problem that we will consider. It was introduced in [6].

Definition 44 Consider the following situation: There are n gossips each of whom has a rumor. An adversary selects a gossip, and invites him to call up any one of the other gossips. However the adversary may not select a gossip who has heard all n rumors, (we suppose that he has found something else to occupy his time). The selected gossip makes a call to one of the other gossips (possibly using randomization to make his choice) and hears all the rumors that the other gossip knows.

The *rumor-spreading problem* is to find a strategy that minimizes the total number of calls made before all the gossips know all the rumors.

This problem is similar to the *gossip problem* [27]. In that problem there are also n gossips, each with a rumor. However in the gossip problem the communication schedule is chosen by the designer of the algorithm; there is no adversary scheduler.

4.1.3 The cooperative collect

Both the asynchronous diagnosis problem and the rumor-spreading problem are closely related to the cooperative collect problem. This problem was first identified as a primitive operation by Saks, Shavit and Woll [49]. It naturally arises in a situation where there are n processes which must learn the value contained in each of n registers.

This primitive is important because it occurs in many of the basic algorithms in the wait-free shared memory model of the distributed-computing literature. It is used in algorithms to achieve consensus, snapshots, coin flipping, bounded round numbers, timestamps and multi-writer registers [1,3,4,5,8,21,24,25,26,28,33,38,53]. It is needed whenever an algorithm requires a process to poll all of the other processes in order to come to some conclusion. For many of these algorithms the time to perform the collect may dominate the running time of the algorithm. So an efficient collect primitive will also improve the efficiency of all these algorithms.

The collect problem is defined as follows:

- There are n processes in a shared memory system. Each process owns a single-writer multi-reader register. A register may be read by any process, but only its owner may write to it.

- Time is modeled asynchronously. The processes are under the control of an adversary scheduler. From time to time the adversary may preempt the current process in favor of another process.

- Each process has a piece of information that we shall refer to as its *item* which it stores in its register. A process performs a *collect* if it learns all the n items. If several processes are performing a collect simultaneously they may cooperate. If a process which has learnt several items writes them on its own register, another process which reads the register will learn several items while only performing a single read operation.

- The cost of a collect is the number of read operations that are performed by processes doing the collect. We do not count reads made by a process after it has finished the collect.

As with the distributed fault diagnosis problem there are several variants of the basic model.

Definition 45 In the *single-shot collect* problem the item associated with each process is fixed. All the processes perform a collect, after which they halt. Our goal is to minimize the number of reads that occur.

In the *repeated collect* problem the item associated with each process may change. From time to time a process may decide to perform a collect, in which case it wishes to obtain a fresh item from every process. Since it is always possible that a item has changed since it was last read, we will define a item to be *fresh* if it was present in the corresponding register at some point during the collect operation.

The single-shot collect is similar to static diagnosis, the repeated collect is similar to dynamic diagnosis. The terms *collect problem* or *cooperative collect* should both be taken to refer to the single-shot collect problem.

The essential difference between the rumor-spreading problem and the collect problem is that in the collect problem the operations of choosing a process to read, reading its register, and adding the information obtained to one's own register, do not take place as a single atomic action. Thus the rumor-spreading problem can be viewed as a simplification of the collect problem.

Many of the results contained in this chapter also appear in [6] where they are presented as results about the repeated cooperative collect problem.

4.1.4 The adversary

The adversary scheduler used in the asynchronous problems is nearly omniscient. We assume that it has complete knowledge about the current state of the system, and that it knows which algorithm every processor is using. But it does not have advance knowledge of the values that will be returned by a random number generator.

There are two variants of the asynchronous models that differ on the strength of their adversary. In the *strong* adversary model the adversary is permitted to preempt a processor at any time. In the *weak* adversary model a processor generates a random number and writes it to its register as a single atomic operation. The weak adversary may preempt a processor at any other time.

Note that we consider testing a processor and reading its register to be an atomic action. We will assume that neither adversary model is able to fool a tester into accepting false information from a register by corrupting the testee after it has been tested but before its register is read. This assumption is justified by supposing that reading the register is part of the testing operation. This assumption is usual is distributed fault diagnosis, for example see [34,17].

Under both adversaries, the adversary can halt a processor after it generates a random number, but before it takes any action as a result of the number it generated. But the weak adversary cannot prevent other processors from discovering the value that was generated.

A weak adversary appears frequently in early work on consensus; it is the "weak model" of Abrahamson [1] and it is used in the consensus paper of Chor, Israeli, and Li [21]. Chandra [20] shows an algorithm which achieves consensus in polylog expected steps against a weak adversary; all known algorithms for consensus need at least linear expected steps against a strong adversary. In general, the weak model enables much better algorithms than the strong model. In some cases a problem may be solved in the weak model, but not in the strong one.

Another way of viewing the weak model, is to claim that the tester has access to the testee's internal state. In distributed computing the model can be justified by claiming that the adversary is modeling faults in the system. Such faults could be modified in unpredictable ways by the actions of a processor (e.g. page fault on trying to access some memory location.) However the value being generated is not in itself likely to cause a halt. The halt only occurs when the processor tries to act on the value generated.

All the results that we will state using an asynchronous model will be against a weak adversary. It is not clear if the results can be extended to work against a strong adversary.

4.2 Spreading Rumors

The rumor-spreading problem is similar to the asynchronous static diagnosis problem. However it admits to a simpler solution, and is easier to analyze. Recall that the rumor-spreading problem involves n gossips spreading n rumors by phone, where the adversary chooses who makes the call, the caller chooses which gossip to contact, and rumors are only passed from the callee to the caller.

We have a straight forward upper bound for the rumor-spreading problem:

Lemma 46 *The rumor-spreading problem can be solved with $n(n-1)$ calls.*

Proof: Each gossip follows the same algorithm: call every other gossip and directly receive his rumor. Thus each person makes $n-1$ calls. □

The upper bound from Lemma 46 can be significantly improved. Clearly no strategy can solve the rumor-spreading problem in fewer than n calls, since each person must make at least one call. It is not hard to show [2] that if everyone uses a deterministic algorithm then the adversary can choose a schedule so that $\Omega(n \log_2 n)$ calls are made.

Now consider Algorithm 4.1. This algorithm gives a deceptively simple strategy for each gossip to follow. When a gossip a makes a move, he calls a gossip b chosen uniformly at random from the set of n gossips. (It is possible that $b = a$.)

Intuitively it seems unlikely that this is the best algorithm. For example, if a has obtained the rumor from $n-1$ gossips, it is clear that a should call the sole gossip whose rumor a does not already possess. Also if $b = a$ then no information can possibly be gained. But this algorithm has the great advantage that it is impossible for the adversary to bias a's selection of b. This makes it much easier to analyze the performance of this algorithm than it otherwise might be.

Definition 47 Let K_s^a be the set of rumors possessed by gossip a at time s. We will say that a gossip a *knows* a set of rumors S at time s when $S \subseteq K_s^a$. Let V_S^s be the

Each gossip follows the same algorithm.
Input: The system of n gossips.

Output: A set of n rumors, one for each gossip in the system.

Method:

```
WHILE (NOT all rumors known)
        CALL a gossip selected at random.
STOP.
```

Algorithm 4.1: Rumor spreading

set of gossips who know the set of rumors S at time s. That is

$$V_S^s = \{a : S \subseteq K_s^a\}.$$

The effect of a calling b at time s is to set K_{s+1}^a to $K_s^a \cup K_s^b$.

Let us look at some set of rumors S and consider how they spread among the gossips. For each S, we can classify the calls into two classes:

- Calls made by gossips that already know all of S. We will call these calls *unproductive* (with respect to S).
- Calls made by gossips that do not already know all of S. We will call these calls *productive* (also with respect to S).

Where it will not cause confusion we will omit a specific reference to S. Note however that a call might be unproductive with respect to some S but productive with respect to a different set S'.

The following lemma shows that, with high probability, the set of rumors, S, known by a single gossip spreads to all the gossips after $O(n \ln n)$ productive calls with respect to S:

Lemma 48 *Fix a starting time s and let $S = K_s^a$. Let T be the number of productive calls after time s before every gossip knows S and let $k > 0$. Then for sufficiently large n (depends on k) we have*

$$Pr[T \geq kn \ln n] \leq \frac{1}{n^{k-3}}$$

Proof: If r gossips know S prior to a productive call, then the probability that $r+1$ gossips know S after the call is r/n. Thus the total waiting time T is given by the sum of a set of independent, geometrically distributed random variables $T_1, T_2, \ldots, T_{n-1}$ with expectations $n, n/2, \ldots, n/(n-1)$. This gives a total expected time of $n \sum_{i=1}^{n-1} \frac{1}{i}$ which is approximately $n \ln n$. However, we wish to establish a stronger claim, by bounding the tail of this sum's distribution. We do this by using moment generating functions. Let $t > 2n - 2$ and define d and c by

$$d = \frac{n}{n-1} \cdot \frac{t-n+1}{t-n+2} \quad \text{and} \quad c = \ln d.$$

The lower bound on t ensures that $d > 1$ and so $c > 0$. Because $c > 0$ we have by Markov's inequality that

$$\Pr[T \geq t] = \Pr\left[e^{cT} \geq e^{ct}\right] \leq \frac{\mathrm{E}\left[e^{cT}\right]}{e^{ct}}.$$

Since the T_i are independent

$$\mathrm{E}\left[e^{cT}\right] = \mathrm{E}\left[\prod_{i=1}^{n-1} e^{cT_i}\right] = \prod_{i=1}^{n-1} \mathrm{E}\left[e^{cT_i}\right].$$

We can evaluate $\mathrm{E}\left[e^{cT_i}\right]$ directly. Let $p = i/n$ and $q = 1 - i/n$. Because $qe^c = qd < 1$ we get,

$$\mathrm{E}\left[e^{cT_i}\right] = p \sum_{j=1}^{\infty} q^{j-1} e^{cj} = pe^c \sum_{j=0}^{\infty} (qe^c)^j = \frac{pe^c}{1-qe^c} = \frac{pd}{1-qd}.$$

Thus

$$\begin{aligned}
\Pr[T \geq t] &\leq \frac{1}{d^t} \prod_{i=1}^{n-1} \frac{\frac{i}{n} d}{1 - (1 - \frac{i}{n})d} \\
&= \frac{1}{d^t} \prod_{i=1}^{n-1} \frac{id}{id + n - dn} \\
&= \frac{(n-1)!}{d^{t-n+1}} \prod_{i=1}^{n-1} \frac{1}{id + n - dn}.
\end{aligned}$$

Because $1 < d < n/(n-1)$ we have that when $1 < i < n$,

$$\frac{1}{id+n-dn} < \frac{n-1}{in+n(n-1)-n^2} < \frac{1}{i-1}.$$

Hence

$$\begin{aligned} \Pr[T \geq t] &\leq \frac{(n-1)!}{d^{t-n+1}} \cdot \frac{1}{d+n-dn} \cdot \frac{1}{(n-2)!} \\ &= \frac{n-1}{d^{t-n+1}(d+n-dn)}. \end{aligned}$$

Let $s = t-n+1$. Then

$$\Pr[T \geq t] \leq \frac{(n-1)d^{-s}}{n-d(n-1)}$$

Let $\lambda = s/(n \ln n)$. Then

$$\begin{aligned} \Pr[T \geq t] &\leq \frac{n-1}{n}\left(\frac{n-1}{n}\right)^{\lambda n \ln n}\left(\frac{s+1}{s}\right)^{s}(s+1) \\ &\leq n^{-\lambda}e(\lambda n \ln n + 1). \end{aligned}$$

Now let $t = kn \ln n$ for k some positive constant. Then $k \geq \lambda \geq k-1$. Assuming that n is large enough that $n^2 \geq e(kn \ln n + 1)$ we conclude

$$\Pr[T \geq kn \ln n] \leq \frac{1}{n^{k-3}} \tag{4.1}$$

□

Lemma 48 tells us that with high probability, after $kn \ln n$ productive calls K_s^a will be known by everybody. Thereafter any further calls must be unproductive calls. So if $3kn \ln n$ calls are performed, at least $\frac{2}{3}$ of them must be unproductive. In other words, most of these $3kn \ln n$ calls are made by gossips that already know K_s^a. That this intuition is true simultaneously for all gossips with high probability is captured in the following lemma:

Lemma 49 *Let s be a time and let $t = s + 3kn \ln n$. For any set of gossips A, define $w(A)$ to be the number of calls made by gossips in A between times s and t. Let $S = K_s^a$. Then, for sufficiently large n,*

$$Pr\left[\exists a \;\; w(V_S^t) < 2kn \ln n\right] \leq \frac{1}{n^{k-4}}$$

Proof: The proof works by showing an upper bound on the number of calls *not* made by gossips in V_S^t. Let $\overline{V_S^t}$ be the complement of V_S^t. Since any gossip in $\overline{V_S^t}$ does not know K_s^a at time t, it cannot have known K_s^a at any time before t, and thus all of its calls prior to t are productive moves with respect to K_s^a. Using Lemma 48 we get

$$\Pr\left[w(\overline{V_S^t}) \geq kn\ln n\right] \leq 1/n^{k-3}.$$

Thus:

$$\begin{aligned}
\Pr\left[\exists a\ w(V_S^t) < 2kn\ln n\right] &\leq \sum_{i=0}^{n}\Pr\left[w(V_S^t) < 2kn\ln n\right] \\
&= n\Pr\left[w(\overline{V_S^t}) \geq kn\ln n\right] \\
&\leq \frac{1}{n^{k-4}}
\end{aligned}$$

□

Because it is likely that $V^t_{K_s^a}$ and $V^t_{K_s^b}$ both do at least $\frac{2}{3}$ of the work, it is likely that these sets overlap for any a and b, i.e. that the information known by any *pair* of gossips at time s is known to a *single* gossip at time $t = s + 3kn\ln n$:

Corollary 50 *Using the notation of Lemma 49,*

$$Pr\left[\exists a, b\ s.t.\ V^t_{K_s^a} \cap V^t_{K_s^b} = \emptyset\right] \leq 1/n^{k-4}.$$

Proof: Let a and b be any two gossips and let $V_a = V^t_{K_s^a}$ and $V_b = V^t_{K_s^b}$. Suppose $w(V_a) \geq 2kn\ln n$ and $w(V_b) \geq 2kn\ln n$ Then $w(\overline{V_b}) < kn\ln n$ and so $V_a \not\subseteq \overline{V_b}$ implying that $V_a \cap V_b \neq \emptyset$. By Lemma 49 the probability that the supposition does *not* hold is at most $1/n^{k-4}$. The result follows. □

In particular, suppose at time s there is some set A of r gossips that between them know all the rumors, i.e.

$$\bigcup_{a\in A} K_s^a \supset \bigcup_a K_0^a.$$

Then at time $s + 3kn\ln n$ there will be a set of $\lceil r/2\rceil$ gossips that between them know all the rumors. Initially the smallest set of gossips that know all the rumors has size n. Therefore after at most $1 + \log_2 n$ intervals of length $3kn \ln n$ there will be a single gossip that knows all of the rumors, i.e. he will have completed his task.

A gossip who has finished will not make any further calls. So all calls made after a gossip has completed are necessarily made by gossips that have not completed. So these calls are productive with respect to knowledge of all the rumors. So applying Lemma 48 shows that after $kn \ln n$ additional moves everybody will know everything with high probability. In summary we have the following:

Theorem 51 *Let k be some constant, and let the adversary and gossips behave as described earlier in this section. Let $c = 3(\log_2 e + 1) = 7.32\cdots$. Then for sufficiently large n the probability that there is somebody who does not know some rumor after $ckn \ln^2 n$ calls is at most $\frac{1}{n^{k-5}}$.*

4.3 The Collect Problem

In this section we will study the (single-shot) collect problem. Recall that in the collect problem there are n processes each of which wishes to collect an item from each of n registers. The distinction between the collect problem and the rumor-spreading problem is that in the former the adversary may preempt a process after it has decided which register to read, but before it has made the read. In the latter the adversary must select the next gossip to make a call before the adversary knows who the gossip will decide to contact. As mentioned in Section 4.1.4, we are using a weak adversary. This means that a process is permitted to generate a random number and write it to its register as an atomic action.

Overall, the approach will be similar to that taken for the rumor-spreading problem. However Algorithm 4.1 (choosing the process to read at random) is not effective against the more powerful adversary used in the collect problem. An adversary strategy that defeats this simple algorithm is to choose one of the registers to be a "poison pill"; any process that attempts to read this register will be halted before it can carry

Each process follows the same algorithm. In the algorithm, we assume that each process stores in its register both the set of items S it has collected so far and its *successor*, the process it selected to read from most recently.

Input: The system of n processes. An integer λ. Each process has an associated register that initially shows its item.

Output: A set of n items, one for each register.

Method:

```
WHILE (NOT all items are known)
        SET p to a random process, and write out p as our successor.
            (We will call this the selection step).
        REPEAT λ ln n times:
                READ the register of p.
                SET S to be the union of S and the items field of p.
                SET p to the successor of the process just read.
                WRITE out the new S and p values.
STOP.
```

Algorithm 4.2: Single-shot collect

out the read. The collect cannot be completed while one register has not been read. Since on average only one read out of every n would attempt to read the poisonous register, $\Omega(n^2)$ reads would be made before the adversary would be forced to let some process actually swallow the poison pill.

We avoid this problem by having each process use Algorithm 4.2. The essential idea is that before attempting to read a register, a process will leave a note saying where it is going; poison pills can thus be detected easily by the trail of corpses leading in their direction. The distance that a process will pursue this trail will be $\lambda \ln n$, where λ is constant chosen to guarantee that the process reaches its target with high probability.

We would like to prove an analogue of Lemma 49 for this more sophisticated algorithm. Lemma 49 does not hold for the collect problem because if S is a set of items, the adversary can prevent S from being spread by halting processes that are about to learn S.

Instead of using the V_S^t sets from Definition 47 we will define analogous sets, U_S^t, for the collect problem. U_S^t is a set of the processes which would learn S, if the adversary permitted them to carry out their next $\lambda \ln n$ reads. More formally:

Definition 52 Let S be a set of items, and let s and t be times with $t \geq s$. Define U_S^t recursively as follows. Let $U_S^s = V_S^s$. If at time t, a process b executes a selection step and chooses a process in U_S^t, then $U_S^{t+1} = U_S^t \cup \{b\}$; otherwise $U_S^{t+1} = U_S^t$.

Note that unlike V_S^t the definition of U_S^t depends on a starting time s. The value to take for s will be clear from the context.

Observe that the sets U_S^t are built up by exactly the same random process as the sets V_S^t, and so we can use Lemma 48 to prove a high-probability bound on how many times the selection step can be executed by a process not already in U_S. This bound translates into a bound on the number of reads because the number of reads executed by any process is at most $\lambda \ln n + 1$ times the number of times it executes the body of the outer loop, i.e., the number of times the selection step is executed.

Unfortunately this is not sufficient to solve the problem. We also need to show that when a process $b \in U_S$ is permitted to continue it will eventually discover S. When b restarts it will read its original target process. It will then follow the trail of successor fields until it has read $\lambda \ln n$ processes in all. It will succeed if it reads $a \in V_S^s$, where s is the start time, which by the construction of U_S will lie at the end of the trial of successor fields. But if the trial is too long, b might give up before it has reached its target. We will show that with high probability this does not happen.

To show this fact we view U_S^t as a collection of rooted trees, one tree for each node $a \in V_S^s$. To simplify the discussion we will suppose that there is exactly one tree, i.e. $V_S^s = \{a\}$. Clearly this will make no difference because if the nodes are divided among several trees, then each individual tree will have a smaller depth than a single large tree would have had.

As each new node c is added to U_S it must select one of the processes b already in U_S; in this case we draw an edge between c and b. Notice that (conditioning on the fact that c selects a process already in U_S) the process b is chosen uniformly from the nodes already in U_S. In Section 4.3.1 we investigate the random variable M_x, which is defined to be the depth of a tree containing $x+1$ nodes generated in precisely this fashion. We prove (equation (4.10)):

Lemma 53 *Let $\lambda \geq 2$, then*

$$Pr[M_{x-1} \geq \lambda \ln x] \leq \frac{1}{x^{\lambda \ln \lambda - \lambda - 1}}$$

Intuitively, the depth of the tree is likely to be bounded by the logarithm of its size because on average the i-th node to be added to the tree will choose as its parent the $(i/2)$-th node. The importance of bounding the depth of the tree is that it gives an immediate bound on the length of a trail that any process in U_S must follow to learn S:

Lemma 54 *Suppose that the depth of the U_S tree does not exceed $\lambda \ln n$. Let b be a process that has completed the inner loop following its first selection of a process in U_S. Then b knows S.*

Proof: The result follows by induction on the size of U_S. The base case is immediate. If b is a process newly added to U_S, either b successfully follows a chain of successor edges until it reaches a, the root of the tree, or at some point it follows an edge leaving some process c that is not an edge in U_S. But then c must have chosen a new successor after its entry into U_S and thus must have completed its inner loop following its entry into U_S. It follows by the induction hypothesis that c knows S, and thus b learns it when it reads c's register. $\square$

Now we have the following extension of Lemma 49.

Lemma 55 *Let the adversary, and the algorithms used by the processes be as defined earlier in this section. Fix a starting time s, let $t = s + 3(kn \ln n + n)(\lambda \ln n + 1)$,* *and define $w(A)$ to be the total number of reads executed by processes in A between s and t. Let $V_a = V^t_{K^a_s}$, $U_a = U^t_{K^a_s}$. Then*

$$Pr[\exists a \ w(V_a) \leq 2(kn \ln n + n)(\lambda \ln n + 1)] \leq \frac{1}{n^{k-4}} + \frac{1}{n^{\lambda \ln \lambda - \lambda - 2}}$$

Proof: We use an argument similar to that used for Lemma 49. Suppose that for some a, $w(\overline{V_a}) \geq (kn \ln n + n)(\lambda \ln n + 1)$. By Lemma 48 after $(kn \ln n)(\lambda \ln n + 1)$ reads every process from $\overline{V_a}$ is in U_a. By lemmas 53 and 54 a process in U_a can do at most $\lambda \ln n + 1$ reads before it follows its trial back and discovers K^a_s. Since there are at most n processes in $\overline{V_a}$, by the pigeonhole principle, at least one of them must have followed its trial back when the set collectively carries out $n(\lambda \ln n + 1)$ reads. Thus a process in $\overline{V_a}$ knows K^a_s. This is a contradiction on the definition of $\overline{V_a}$.

Each of the lemmas used has an associated upper bound on the probability that the event it describes does not occur. Summing these probabilities gives the right hand side of the statement of this lemma. □

This lemma can be used in exactly the same way as in section 4.2 to prove the following theorem:

Theorem 56 *Let k, λ be constants, $k \geq 1$, $\lambda \geq 2$, and let the adversary and processes behave as described earlier in this section. Assume that $n \geq 3$ and let $c = 26$. Then the probability that the collect is incomplete after $c\lambda kn \ln^3 n$ reads is at most*

$$\frac{1}{n^{k-5}} + \frac{1}{n^{\lambda \ln \lambda - \lambda - 3}}.$$

Proof: The argument is essentially the same as used for Theorem 51. The resulting cost is given by

$$3(kn \ln n + n)(\lambda \ln n + 1)(\log_2 n + 1)$$

which is at most $26k\lambda n \ln^3 n$ under the assumptions (needed for the lemmas) that $k \geq 1$ and $\lambda \geq 2$, and the further assumption that $n \geq 3 > e$ (implying $\ln^3 n > \ln^2 n > \ln n$). □

In particular if we take $k = \lambda \geq 9$ we can combine the terms in the probability bound to get as a special case that the probability that the cooperative collect is incomplete after $ck^2 n \ln^3 n$ reads is at most $\frac{2}{n^{k-5}}$ (where $c = 26$ as in the theorem).

The best previous result for the cooperative collect problem against a weak adversary is $O(n^{3/2})$ reads which was shown by Ajtai, Aspnes, Dwork, and Waarts in [2]. Our $O(n \ln^3 n)$ result compares very favorably with this result. In [49] Saks, Shavit and Woll obtain a $O(n \ln n)$ result, but they are using a different model with a weaker adversary which is required to make a constant fraction of the processes behave in a synchronous manner.

4.3.1 Proof of Lemma 53

In this section we investigate the expected depth of a rooted tree which is built by adjoining each new vertex to one of the existing vertices chosen at random. We will show that with high probability the depth of the tree of i vertices is $O(\ln i)$.

An example of a tree generated in this fashion is given in figure 4.3. The numbers drawn in the nodes show the order in which the nodes were adjoined. Observe that the tree tends to become much broader than it is deep.

Let T_i be a random variable whose value is a rooted tree with $i+1$ vertices, including the root vertex. So T_0 consists of the root vertex only. Let T_{i+1} be defined by uniformly selecting one of the $i+1$ vertices in T_i and attaching a new vertex to the selected vertex.

Define random variables D_i to be the depth of the ith vertex, where the root has depth -1, a vertex adjacent to the root has depth 0 and so on. Let M_i be the depth of the tree T_i, so

$$M_i = \max_{j \leq i} D_j.$$

Now define indicator variables for $i \geq 0$, $d \geq -1$,

$$X_{i\,d} = \begin{cases} 1 & \text{if } D_i = d \\ 0 & \text{otherwise} \end{cases}$$

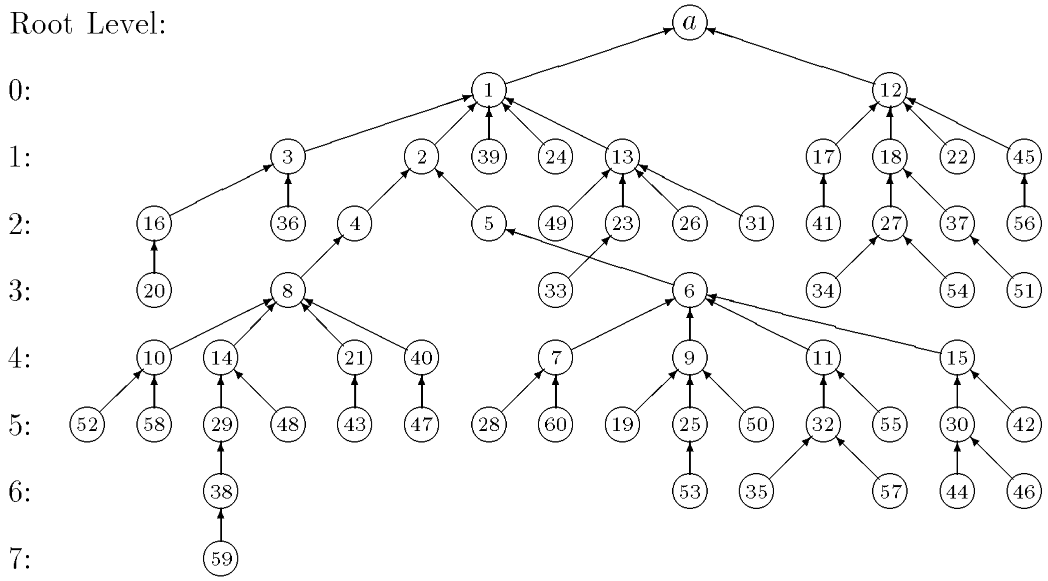


Figure 4.3: Example of a U_S tree with 60 nodes

Let $x_{i\,d} = \Pr[D_i = d] = \Pr[X_{i\,d} = 1] = \mathrm{E}[X_{i\,d}]$.

From the construction of the tree we have for $i \geq 1$ and $d \geq 0$

$$\Pr[X_{i\,d} = 1] = \frac{1}{i} \sum_{j=0}^{i-1} X_{j\,d-1}.$$

Taking expectations we get

$$\mathrm{E}[X_{i\,d}] = \frac{1}{i} \sum_{j=0}^{i-1} \mathrm{E}[X_{j\,d-1}].$$

So the $x_{i\,d}$ are defined by the recurrence equation

$$x_{i\,d} = \begin{cases} \frac{1}{i} \sum_{j=0}^{i-1} x_{j\,d-1} & \text{if } i \geq 1 \text{ and } d \geq 0 \\ 1 & \text{if } i = 0 \text{ and } d = -1 \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

From (4.2) we can derive two further recurrence equations, for $i \geq 1$, $d \geq 0$

$$x_{i\,d} = \frac{i-1}{i} x_{i-1\,d} + \frac{1}{i} x_{i-1\,d-1} \tag{4.3}$$

$$\text{and} \quad x_{i\,d} = \frac{1}{i} \sum_{0<i_1<i_2<\ldots<i_d<i} \prod_{j=1}^{d} \frac{1}{i_j}. \tag{4.4}$$

Now we can use (4.3) to find the expectation of D_i, since

$$\begin{aligned}
\mathrm{E}\left[D_i\right] &= \sum_{d=0}^{\infty} d x_{i\,d} = \sum_{0}^{\infty} d\left(\frac{i-1}{i} x_{i-1\,d} + \frac{1}{i} x_{i-1\,d-1}\right) \\
&= \frac{i-1}{i}\sum_{0}^{\infty} d x_{i-1\,d} + \frac{1}{i}\sum_{1}^{\infty}(d-1) x_{i-1\,d-1} + \frac{1}{i}\sum_{1}^{\infty} x_{i-1\,d-1} \\
&= \frac{i-1}{i}\mathrm{E}\left[D_{i-1}\right] + \frac{1}{i}\mathrm{E}\left[D_{i-1}\right] + \frac{1}{i}\cdot 1 \\
&= \mathrm{E}\left[D_{i-1}\right] + \frac{1}{i}
\end{aligned}$$

Since $\mathrm{E}\left[D_0\right] = -1$ we get

$$\mathrm{E}\left[D_i\right] = \sum_{j=2}^{i} \frac{1}{j} \leq \ln i \tag{4.5}$$

This shows that in a tree with r vertices the expected depth of any particular vertex is at most $\ln r$, which suggests that the expected depth of the entire tree is also of the order of $\ln r$. To prove this we will need to get an upper bound on $x_{i\,d}$.

By comparing the identity

$$\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d \equiv \sum_{i_1=1}^{i-1}\sum_{i_2=1}^{i-1}\cdots\sum_{i_d=1}^{i-1}\prod_{j=1}^{d}\frac{1}{i_j}$$

with (4.4) we see that

$$\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d = i x_{i\,d} d! + \text{terms involving squares.} \tag{4.6}$$

Hence

$$x_{i\,d} \leq \frac{\left(\sum_{j=1}^{i-1}\frac{1}{j}\right)^d}{i\cdot d!} \leq \frac{(1+\ln(i-1))^d}{i\cdot d!} \tag{4.7}$$

In fact we can show that as $i \to \infty$, $x_{i\,d} \to \ln^d i/(id!)$. That is, the D_i are asymptotically Poisson distributed with parameter $\ln i$.

Let $h = d/\ln i$. Then using Stirling's formula we have

$$\frac{(1+\ln i)^d}{d!} \quad = \quad \left(\frac{d}{h}\right)^d \frac{\left(1+\frac{h}{d}\right)^d}{d!} \leq 2\left(\frac{d}{h}\right)^d \frac{e^h}{d!}$$

$$\begin{aligned} &\leq \frac{2e^h}{\sqrt{2\pi d}}\left(\frac{d}{h}\right)^d\left(\frac{e}{d}\right)^d \leq e^h e^{h(1-\ln h)\ln i} \\ &\leq \frac{1}{i^{h\ln h-h-1}} \end{aligned} \tag{4.8}$$

assuming that $i \geq 3$. Let $k \geq 2$. By combining (4.7) and (4.8) we obtain

$$x_{i\,d} \leq \frac{1}{i^{k\ln k-k}} \quad \text{provided } i \geq 3 \text{ and } d \geq k\ln i \tag{4.9}$$

Suppose $M_x \geq y$ for some x and y. If there is a node with depth bigger than y there must be a node of depth exactly y. Thus using (4.2) we have that

$$\Pr\left[M_x \geq y\right] \leq \sum_{i\leq x}\Pr\left[D_i = y\right] = \sum_{i\leq x} x_{i\,y} = (x+1)x_{x+1\,y+1}.$$

So by applying (4.9) we can conclude since $k \geq 2$

$$\Pr\left[M_{x-1} \geq k\ln x\right] \leq \frac{1}{x^{k\ln k-k-1}} \tag{4.10}$$

In particular if $k \geq 9$ we have that $k\ln k - k - 1 \geq k$ so

$$\Pr\left[M_{x-1} \geq k\ln x\right] \leq \frac{1}{x^k} \quad \text{for } k \geq 9. \tag{4.11}$$

4.3.2 The adversary's power

Recall from Section 4.1.4 that we are using a weak adversary model in which we suppose that a process is able to generate a random number and write it to its register in a single step. The adversary may not choose to halt a process after it has selected its next target, but before it has made its selection visible to other processes. It is not clear if the collect problem can still be solved in $O(n\ln^r n)$ reads if the strong model is used, i.e. if we remove this restriction from the adversary.

The collect algorithm (Algorithm 4.2) could take $\Omega(n^2)$ reads to finish against a strong adversary. A strong adversary strategy that defeats this algorithm is to choose one of the registers to be a "poison pill"; any process that decides to read this register will be halted before it can write out its intention to read the register. The collect cannot be completed while one register has not been read. Since on average only one selection step out of every n would choose the poisonous register, $\Omega(n^2)$ reads would be made before the adversary would be forced to let some process actually swallow the poison pill.

We would like to obtain an $O(n \ln^r n)$ algorithm against a strong adversary. Unfortunately the proof techniques considered in this chapter are not powerful enough to obtain this. All we have assumed about each item is that initially it is only known to one of the processes. We have never used the fact that the items are initially distributed so that each process knows exactly one item. Thus our analysis does not distinguish between the cooperative collect and the library collect which is defined below.

Definition 57 The *library collect* problem is a variant of the single-shot cooperative collect problem. There are n processes, and n items. One process, chosen by the adversary, is designated to be the *library* process. The other processes are not told which process is the library process. Initially the library process knows all the items; all of the other processes know none of the items. In all other respects the library collect problem is the same as the single-shot collect problem.

Any algorithm for the library collect problem in which processes only record which items they have discovered, can be defeated by a a strong adversary. It uses the same poison-pill argument, with the library processor as the pill. None of the other processors are able to gain any information at all.

We need an algorithm for the cooperative collect which takes account of the initial distribution of the items among the registers. The algorithm used by Saks, Shavit and Woll [49] is a good candidate. In this algorithm a process alternates between reading a register selected at random from all the registers, and one selected at random from among those registers whose items it has not discovered.

4.3.3 Asynchronous distributed diagnosis

Asynchronous distributed diagnosis is very similar to the collect problem. We may view it as a special case of the collect problem in which the items of knowledge to be collected consist of the status of each processor. A slight additional complexity is introduced because only the good processors perform the diagnosis. The equivalence is shown by the following lemma:

Lemma 58 *Suppose there is an algorithm $\mathcal{A}$ that can solve the single-shot collect problem for n processes with $f(n)$ reads with some probability p regardless of the adversary's strategy. Then there is an algorithm $\mathcal{B}$ for the distributed diagnosis problem with n processors that will finish after at most $f(n)$ tests with the same probability p.*

Proof: Let $\mathcal{A}$ be an algorithm for the single-shot collect problem with the required properties. If we restrict the behavior of an adversary used in the collect problem then algorithm $\mathcal{A}$ will still finish in at most $f(n)$ reads against the restricted adversary. Consider a class of adversaries which will only schedule processes from a particular set G until every process in G knows all the items, after which point they will schedule the remaining processes. Clearly, against this adversary every process in G will know all the items within $f(n)$ reads with probability p.

We construct from $\mathcal{A}$ an algorithm $\mathcal{B}$ for the distributed diagnosis problem. Replace each occurrence of "process i reads process j's register" with "processor i tests processor j". As long as the testers are all good then the spread of knowledge of the status of a processor under algorithm $\mathcal{B}$ will duplicate the spread of knowledge of the item of the corresponding process under algorithm $\mathcal{A}$.

When measuring the cost of $\mathcal{B}$ against an adversary we do not count tests performed by faulty processors. Also the actions of the faulty processors do not affect the behavior of the good processors. So the cost of the diagnosis would be the same if we modified the adversary so that it only scheduled faulty processors after all the good processors have completed their diagnosis. Thus we may assume that $\mathcal{B}$ is only performed against adversaries which do not schedule faulty processors until after all the good processors have finished their diagnosis.

Equivalently we are considering adversaries with $G = \{i : \text{processor } i \text{ is good}\}$. Against these adversaries algorithm $\mathcal{B}$ will use the same number of tests by good processors as algorithm $\mathcal{A}$ uses reads by processes in G. Thus algorithm $\mathcal{B}$ will be finished in $\leq f(n)$ tests with probability p. □

When all the processors are good, we take $G = \{1, 2, \ldots, n\}$. In this case the diagnosis uses as many tests, as the collect uses reads, against any given scheduler. So Lemma 58 cannot be improved.

Using Lemma 58 we may immediately obtain a corollary of Theorem 56 which states the same times bounds for distributed diagnosis as Theorem 56 gave for the cooperative collect.

Corollary 59 *Let k, λ be constants, $k \geq 1$, $\lambda \geq 2$. Perform distributed diagnosis on n processors, in which each processor uses an algorithm analogous to Algorithm 4.2. Assume that $n \geq 3$ and let $c = 26$. Then the probability that the diagnosis is incomplete after $c\lambda kn \ln^3 n$ tests is at most*

$$\frac{1}{n^{k-5}} + \frac{1}{n^{\lambda \ln \lambda - \lambda - 3}}.$$

4.4 Repeated Collect

In this section we wish to study the repeated collect problem. Recall from Section 4.1.3 that in the repeated collect problem there are n processes, each with its own register. This problem differs from the single-shot collect problem in that the item stored in each register may change. From time to time a process may perform a collect, in which it reads a *fresh* item from each register. An item is defined to be fresh with respect to a particular collect if it was present in the corresponding register at some point during the collect operation.

We will show how to construct an algorithm for asynchronous dynamic distributed diagnosis from an algorithm for the repeated collect problem. We will discuss ways of measuring the performance of these algorithms, and state some competitive-ratio results.

4.4.1 Timestamps

We use a timestamp algorithm to ensure that we obtain a fresh item from each register. Whenever a process starts a new collect operation it generates a new timestamp. In its internal table it stores the most recent item that it has seen from each register. Each stored item is tagged with a timestamp from every process, namely the most recent timestamp that was known to have been generated by the process when the item was first read. When an item is directly read, (i.e. read from the register of the process whose item it is) the item is tagged with the most recent timestamps that are known to the reader. When an item is indirectly read, the timestamps it is tagged with are copied over with the item. A process knows that an item is fresh with respect to its current collect, if the item is tagged with either the timestamp that the process generated when it started the collect, or with a later timestamp.

Other than the timestamps the algorithm is the same as Algorithm 4.2, the algorithm for the single-shot collect. For variety, instead of presenting the algorithm for repeated collects we present Algorithm 4.4 which is an algorithm for the asynchronous dynamic distributed diagnosis problem. This problem is analogous to the repeated-collect problem in the same way that the static distributed diagnosis problem is analogous to the single-shot collect problem.

In order to measure the performance of the repeated-collect algorithm, we will prove an upper bound on the number of reads needed to complete all the collect operations that are in progress at any particular time. This is called the *collective latency* [2] of the algorithm.

Theorem 60 *Fix a starting time s. Let k, λ, n, and c be as in Theorem 56. Each* *process carries out a certain number of reads between s and the time at which it completes the collect it was working on at time s. Let T be the sum over all processes of these numbers. Then*

$$Pr\left[T > 2c\lambda kn\ln^3 n\right] \leq 2\left(\frac{1}{n^{k-5}} + \frac{1}{n^{\lambda\ln\lambda-\lambda-3}}\right)$$

Proof: Each read that contributes to T must be in one of the following two classes:

1. The reader does not know all the timestamps that were current at time s.

2. The reader knows all n of these timestamps.

Observe that in making the class (1) reads the system is behaving exactly like the collect problem, where the information being spread throughout the system is the latest timestamp for each process. So Theorem 56 gives an upper bound on the number of class (1) reads that are performed.

Similarly class (2) reads behave like an instance of the collect problem in which the information being spread is an item for each process which is tagged with timestamps from each process that were valid at time s. Again Theorem 56 gives a bound on the number of reads made.

The result is obtained by doubling the bound on the number of reads given in the theorem. The probability of failure is also doubled. □

4.4.2 Asynchronous dynamic distributed diagnosis

As in Lemma 58 we can convert an algorithm for the repeated collect problem to an algorithm for the dynamic distributed diagnosis problem. Replace each occurrence of "process i reads process j's register" with "processor i tests processor j". As before the diagnosis algorithm will use no more tests by good processors than than the collect algorithm used reads by the corresponding processes. Algorithm 4.4 gives our proposed algorithm for the diagnosis problem.

All good processors use the same algorithm, but on each processor q is set to the processor's own index number.
Input: A system of n processors, an integer parameter λ, an (n)-dimensional table σ of statuses, and an (n,n)-dimensional table τ of timestamps.

Output: A fresh diagnosis of the system.

Method: Let σ_i be the most recent status obtained from processor i. Let τ_{ij} be the timestamp from processor j that is tagged to σ_i.

```
SET σ_q to good.
SET τ_qq to a fresh timestamp t, greater than all previous timestamps.
WHILE (∃i τ_iq < t)
        SET p to a random processor.
        REPEAT λ ln n times:
                TEST processor p.
                IF (p is good)
                   FOR i = 1 TO n
                        IF (τ_ii < p[τ_ii]) SET σ_i = p[σ_i], ∀j τ_ij = p[τ_ij].
                SET σ_p to the status of p.
                SET ∀i τ_pi = τ_ii.
                SET p = p[p].
STOP.
```

Algorithm 4.4: Dynamic distributed diagnosis

One of the reasons that many distributed algorithms use a naive collect is because it is hard to prove that the repeated cooperative collect is more efficient. The problem is that the performance of a collect greatly depends on the schedule used by the processes that are running the collect. Some kind of *competitive analysis*[50] is needed. Competitive analysis measures the performance of an on-line algorithm by comparing it with the performance of an optimal off-line algorithm working on the same data. The efficiency of the on-line algorithm is then given as the ratio of the performance of the two algorithms on the worst case data.

In our case we wish to compare the behavior of our algorithm on a given schedule with that of an algorithm which is optimized to use the same schedule. In [6] we show how to use the bound on the collective latency of the repeated-collect algorithm to obtain competitive ratios for it. These ratios will also apply to the distributed diagnosis problem.

Two competitive models are considered. First the competitive latency model of Ajtai et al. [2]. This compares the repeated collect algorithm with an optimal algorithm (called the *champion*) running with the same sequence of requests to carry out a collect and on the same schedule. We obtain the result:

Theorem 61 *The competitive latency of the repeated collect algorithm is* $O(\log_2^3 n)$.

Proof: The details are given in our paper [6]. □

This result holds even against an *adaptive off-line* adversary [14], which is allowed to choose the champion algorithm after seeing a complete execution of the candidate.

The second competitive model is the throughput competitiveness model of Aspnes and Waarts [7]. In this model we compare the number of collects made by the repeated collect algorithm, if each process starts a new collect on the completion of each old collect, with the number of collects made by a champion optimized to run on the same schedule.

Unfortunately, the throughput model does not permit as good a competitive ratio for cooperative collect as the latency model: Aspnes and Waarts give a lower bound of $\Omega(\sqrt{n})$. However, our algorithm come very close to this lower bound. We obtain the result:

Theorem 62 *The competitive throughput of the repeated cooperative collect algorithm is* $O(n^{1/2}\log_2^{3/2} n)$.

Proof: The details are given in our paper [6]. □

The same ratios hold for dynamic distribted diagnosis.

4.5 Synchronous Model

In this section we consider distributed fault-diagnosis models in which the processors are synchronous. As in the previous sections we will approach these models via the rumor-spreading and cooperative-collect problems. In effect we will view synchronous time as a special case of asynchronous time in which instead of there being an adversary scheduler the processors follow a uniform schedule.

We will show that $O(\log^2 n)$ rounds of testing suffice, which corresponds to $O(n \log^2 n)$ individual tests. This result is a slight improvement over the $O(n \log^3 n)$ result obtained in Corollary 59. It is not surprising that we obtain better results when we remove the adversary scheduler, since we can then select an algorithm that is optimized for the fixed schedule.

No processor may participate in more than one test in a round. Since there is no controller in a distributed model, each processor independently decides which processor (if any) to test. This leads to a risk of conflict between two tests when, for example, two processors both select the same processor to test.

There are several reasonable ways to resolve this situation. We will do it by supposing that whenever two tests conflict then neither test is performed. Another possible model would be to suppose that exactly one of the tests succeeds, where the successful test is chosen at random from the conflicting tests. Clearly diagnosis would be completed faster under this model, and so the results of this section would also hold for this model.

For the rumor-spreading problem the gossips follow Algorithm 4.5. This is the same as the usual algorithm in which each gossip randomly selects which person to call, except that each gossip will independently rule himself out unless an event with probability λ occurs. Thus in each round we expect only λn of the gossips will make a call, reducing the risk of conflicts between the calls that are made. The expected number of conflict-free calls is approximately

$$\lambda(1-\lambda)\left(1-\frac{2}{n}\right)^{\lambda n-1} n \approx \lambda(1-\lambda)e^{-2\lambda}n.$$

Input: The system of n gossips, and a probability λ.

Method:

```
WHILE (NOT all rumors known)
        IF (event with probability λ occurs)
           CALL a gossip selected at random.
STOP.
```

Algorithm 4.5: Synchronous rumor spreading

This is maximized by setting $\lambda = 1 - \sqrt{2}/2$, in which case we expect about $(0.12)n$ of the gossips to make a call that doesn't conflict with any of the other calls.

Thus under Algorithm 4.5 we expect a constant fraction of the gossips to make a successful call in each round. So with high probability, every gossip will make at least one call every $O(\log n)$ rounds.

Suppose a rumor is known to less than half of the gossips, and that during a sequence of $O(\log n)$ rounds every gossip makes a call. Then at least half of the calls that are made are productive with respect to the rumor. So by Lemma 48, after $O(\log^2 n)$ rounds each rumor is known to at least half of the gossips with high probability.

Once a rumor is known to at least half of the gossips then each of the gossips that doesn't know the rumor will, with high probability, learn it during its next $O(\log n)$ calls. By using arguments similar to those used earlier in this chapter to estimate the probabilities, we can prove:

Theorem 63 *Let the gossips in the synchronous rumor-spreading problem follow Algorithm 4.5 with $\lambda = (0.293)n$, and let $k > 0$. Then after $O(k \log^2 n)$ rounds of calls have been made, the probability that there is a gossip that does not know all the rumors is $O(n^{-k})$.*

Since there is no adversary scheduler in the synchronous model there is no distinction between the synchronous single-shot collect and the synchronous rumor-spreading problem. So Theorem 63 also applies to the synchronous rumor spreading problem. By a synchronous variant of Lemma 58 we can prove an analogous result for synchronous distributed diagnosis.

4.6 Conclusion

In this chapter we have introduced several models of distributed fault diagnosis. We have discovered that, in a distributed setting, the hard part of the system-diagnosis problem consists of performing a "collect", in which every processor obtains some piece of information from every other processor. Thus we have studied the rumor-spreading and cooperative-collect problems.

It is clear that $\Omega(n)$ operations are needed to perform a collect, and that $O(n^2)$ operations suffice. We have shown that a collect can be performed, with high probability, in $O(n \ln^3 n)$ operations, which is a large improvement on the previous best known result of $O(n^{3/2} \log n)$ operations. Our results are improved slightly if the adversary is restricted. If the adversary is not permitted to preempt a processor after it has selected a target to read, but before it has performed the read, then $O(n \ln^2 n)$ operations will suffice. Also if the system is considered to be synchronous, which can be viewed as a restriction on the normally asynchronous adversary, then $O(n \ln^2 n)$ operations suffice.

Analogous results hold for the distributed diagnosis problem. It is clear that $\Omega(n)$ tests are needed to perform a distributed diagnosis, and that $O(n^2)$ tests suffice. We have shown that a distributed diagnosis can be performed, with high probability, with $O(n \ln^3 n)$ tests. Our results are improved slightly if the adversary is restricted. If the adversary is not permitted to preempt a processor after it has selected a target to test, but before it has performed the test, then $O(n \ln^2 n)$ tests will suffice. Also if the system is considered to be synchronous, which can be viewed as a restriction on the normally asynchronous adversary, then $O(n \ln^2 n)$ tests suffice.

We also considered the repeated collect problem, and presented an algorithm to solve it. We found that the competitive latency of our repeated collect algorithm is $O(\log_2^3 n)$ and that the competitive throughput of the algoirithm is $O(n^{1/2} \log_2^{3/2} n)$. So if we use the first ratio, we are within a log factor of optimality. It is possible to show that no algorithm can give a competitive throughput that is below $O(\sqrt{n})$. Analagous results hold for the dynamic distributed diagnosis algorithm.

Chapter 5

Conclusion

We have investigated several models of fault diagnosis. We have found that different models of diagnosis require quite different algorithms to perform the diagnosis and different techniques to analysis the algorithms.

In Chapter 2 we considered a static model in which each processor has a fixed state and used deterministic algorithms to analyze it. We were able to reduce the number of rounds needed for general diagnosis from 32 to 10. We have also proved that at least 5 rounds of testing are needed. Thus, the general diagnosis problem is almost entirely solved. Since there is some room for improvement in the lower bound analysis, which often fails to distinguish between a testing b and b testing a, it seems likely that the true smallest number of rounds needed to complete diagnosis for all sufficiently large n is closer to 10 than to 5.

In Chapter 3 we considered a dynamic model, in which processors are continuously being corrupted and repaired, at a rate of at most t per round. We came to the surprising conclusion that we could indefinitely limit the number of faulty processors in the system to $O(t \log_2 t)$, a bound which does not depend on n, the size of the system. We were able to prove that this bound is the best possible, for a deterministic algorithm. An interesting open question for this model is whether we obtain a more favorable result if we use randomized algorithms.

In Chapter 4 we considered several distributed models of diagnosis. The main result was to show that static distributed diagnosis could be completed in $O(n \ln^3 n)$ tests, with high probability. This result is proved using a weak model of the adversary's power. It would be very interesting to know if diagnosis can still be completed in $O(n \ln^r n)$ tests against a strong adversary.

Bibliography

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proc. 7th Ann. ACM Symp. Princip. Dist. Comp.*, pages 291–302, 1988.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Ann. Symp. Found. Comput. Sci.*, pages 401–411, 1994.

[3] J. Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3):414–431, May 1993. An earlier version appeared in Proc. 9th Ann. ACM Symp. Princip. Dist. Comp., pp. 325–331, August 1990.

[4] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, Sept. 1990.

[5] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proc. 2nd Ann. ACM Symp. Par. Alg. Arch.*, pages 340–349, 1990.

[6] J. Aspnes and W. Hurwood. Spreading rumors rapidly despite an adversary. In *Proc. 15th Ann. ACM Symp. Princip. Dist. Comp.*, pages 143–151, 1996.

[7] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proc. 28th Ann. ACM Symp. Theor. Comput.*, pages 237–246, 1996.

[8] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log_2 n)$ operations. In *Proc. 12th Ann. ACM Symp. Princip. Dist. Comp.*, pages 29–40, 1993.

[9] A. Bagchi and S. L. Hakimi. An optimal algorithm for distributed system level diagnosis. In *Proc. 21st Ann. IEEE Symp. Fault-Tol. Comp.*, pages 214–221, 1991.

[10] A. Baker. *A Concise Introduction to the Theory of Numbers*. Cambridge University Press, 1984.

[11] R. Beigel, W. Hurwood, and N. Kahale. Fault diagnosis in a flash. In *Proc. 36th Ann. Symp. Found. Comput. Sci.*, pages 571–580, 1995.

[12] R. Beigel, S. R. Kosaraju, and G. F. Sullivan. Locating faults in a constant number of testing rounds. In *Proc. 1st Ann. ACM Symp. Par. Alg. Arch.*, pages 189–198, 1989.

[13] R. Beigel, G. Margulis, and D. A. Spielman. Fault diagnosis in 32 parallel testing rounds. In *Proc. 5th Ann. ACM Symp. Par. Alg. Arch.*, pages 21–29, 1993.

[14] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proc. 22nd Ann. ACM Symp. Theor. Comput.*, pages 379–386, 1990.

[15] C. Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973.

[16] R. P. Bianchini, Jr. and R. Buskens. An adaptive distributed system level-diagnosis algorithm and its implementation. In *Proc. 21st Ann. IEEE Symp. Fault-Tol. Comp.*, pages 222–229, 1991.

[17] R. P. Bianchini, Jr., K. Goodwin, and D. S. Nydick. Practical application and implementation of distributed system-level diagnosis theory. In *Proc. 20th Ann. IEEE Symp. Fault-Tol. Comp.*, pages 332–339, June 1990.

[18] P. M. Blecher. On a logical problem. *Discrete Math.*, 43:107–110, 1983.

[19] D. M. Blough and A. Pelc. Diagnosis and repair in multiprocessor systems. *IEEE Transactions on Computers*, 42(2):205–217, Feb. 1993.

[20] T. D. Chandra. Polylog randomized wait-free consensus. In *Proc. 15th Ann. ACM Symp. Princip. Dist. Comp.*, pages 166–175, 1996.

[21] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th Ann. ACM Symp. Princip. Dist. Comp.*, pages 86–97, 1987.

[22] A. T. Dahbura and G. M. Masson. An $O(n^{2.5})$ fault identification algorithm for diagnosable systems. *IEEE Trans. Comput.*, June 1984.

[23] E. W. Dijkstra. Self stabilization in spite of distributed control. *Commun. ACM*, 17:643–644, 1974.

[24] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible! In *Proc. 21st Ann. ACM Symp. Theor. Comput.*, pages 454–465, 1989. An extended version appears in IBM Research Report RJ 6785, March 1990.

[25] C. Dwork, M. Herlihy, and O. Waarts. Bounded round numbers. In *Proc. 12th Ann. ACM Symp. Princip. Dist. Comp.*, pages 53–64, 1993.

[26] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proc. 24th Ann. ACM Symp. Theor. Comput.*, pages 655–666, 1992.

[27] S. Even and B. Monien. On the number of rounds needed to disseminate information. In *Proc. 1st Ann. ACM Symp. Par. Alg. Arch.*, 1989.

[28] R. Gawlick, N. Lynch, and N. Shavit. Concurrent timestamping made simple. In *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.

[29] S. L. Hakimi and A. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, C-23(1):86–88, Jan. 1974.

[30] S. L. Hakimi and K. Nakajima. On adaptive system diagnosis. *IEEE Transactions on Computers*, C-33(3):234–240, Mar. 1984.

[31] S. L. Hakimi, M. Otsuka, E. F. Schmeichel, and G. F. Sullivan. A parallel fault identification algorithm. *Journal of Algorithms*, 11:231–241, 1990.

[32] S. L. Hakimi and E. F. Schmeichel. An adaptive algorithm for system level diagnosis. *Journal of Algorithms*, 5:526–530, 1984.

[33] M. Herlihy. Randomized wait-free concurrent objects. In *Proc. 10th Ann. ACM Symp. Princip. Dist. Comp.*, Aug. 1991.

[34] S. H. Hosseini, J. G. Kuhl, and S. M. Reddy. A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Transactions on Computers*, C-33(3):223–233, Mar. 1984.

[35] S. Huang, J. Xu, and T. Chen. Characterization and design of sequentially t-diagnosable systems. In *Proc. 19th Ann. IEEE Symp. Fault-Tol. Comp.*, pages 554–559, 1989.

[36] W. Hurwood. Dynamic fault diagnosis. Technical Report 1056, Yale University, 1995.

[37] W. Hurwood. Ongoing diagnosis. In *15th Ann. IEEE Symp. on Reliable Dist. Systems*, 1996. To appear.

[38] A. Israeli and M. Li. Bounded time stamps. In *Proc. 28th Ann. Symp. Found. Comput. Sci.*, 1987.

[39] J. G. Kuhl and S. M. Reddy. Distributed fault-tolerance for large multiprocessor systems. In *7th Ann. Symp. on Comp. Architecture*, pages 23–30, 1980.

[40] J. G. Kuhl and S. M. Reddy. Fault-diagnosis in fully distributed systems. In *Proc. 11th Ann. IEEE Symp. Fault-Tol. Comp.*, pages 100–105, 1981.

[41] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

[42] U. Manber. System diagnosis with repair. *IEEE Transactions on Computers*, C-29(10):934–937, Oct. 1980.

[43] G. A. Margulis. Explicit group-theoretical constructions of combinatorial schemes and their applications to the designs of expanders and concentrators. *Problems of Information Transmission*, 24:39–46, 1988.

[44] K. Nakajima. A new approach to system diagnosis. In *Proc. 19th Annu. Allerton Conf. Commun. Contr. and Comput.*, pages 697–706, Sept. 1981.

[45] E. M. Palmer. *Graphical Evolution*. John Wiley & Sons, New York, 1985.

[46] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16(6):848–853, Dec. 1967.

[47] S. Rangarajan, A. T. Dahbura, and E. A. Ziegler. A distributed system-level diagnosis algorithm for arbitrary network topologies. *IEEE Transactions on Computers*, 44(2):312–334, Feb. 1995.

[48] P. Ribenboim. *The Book of Prime Number Records*. Springer-Verlag, second edition, 1989.

[49] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus — making resilient algorithms run fast in practise. In *Proceedings of 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 351–362, 1991.

[50] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[51] J. Spencer. *Ten Lectures on the Probabilistic Method.* Society for Industrial and Applied Mathematics, 1987.

[52] G. F. Sullivan. A polynomial time algorithm for fault diagnosability. In *Proc. 25th Ann. Symp. Found. Comput. Sci.*, pages 148–156, 1984.

[53] P. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th Ann. Symp. Found. Comput. Sci.*, 1986.

Appendix A

Reduction Table

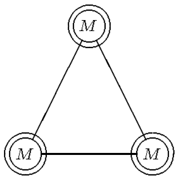
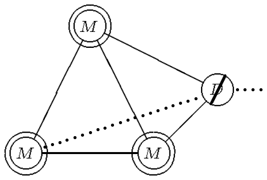
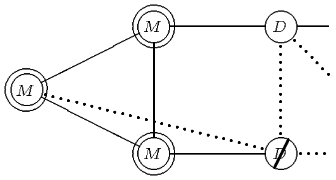
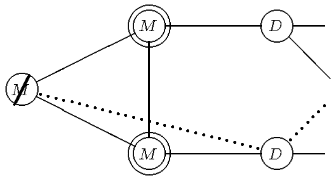
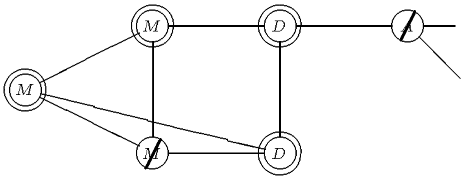
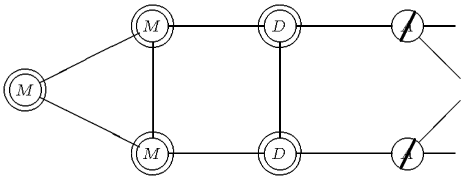
This appendix lists the thirty two graph reductions used in Section 2.4.3. The following notation is used:

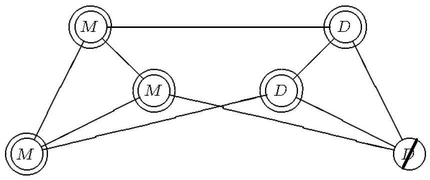
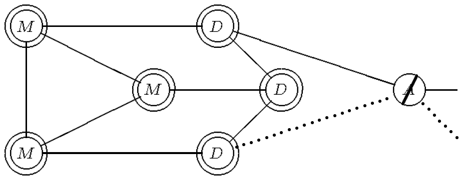
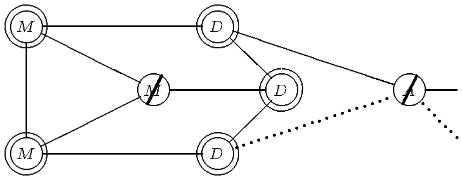
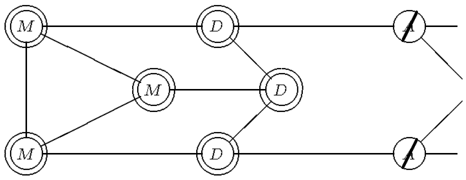
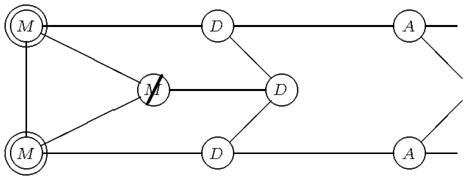
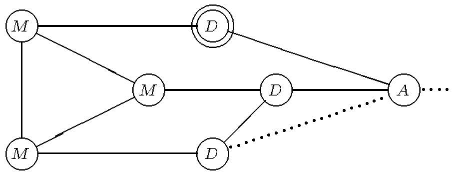
- Every vertex and edge drawn in the diagram is distinct. If no edge is shown connecting to a vertex then there isn't an edge there, even if the vertex only has degree 2 in the diagram.

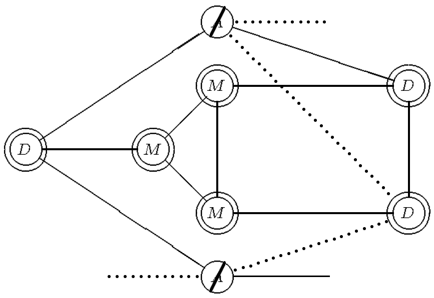
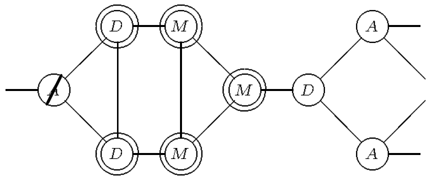
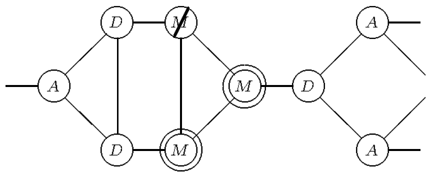
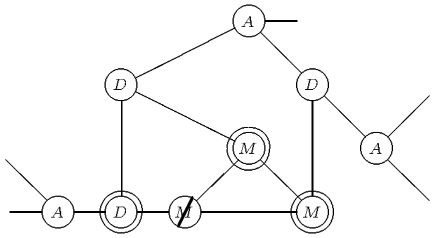
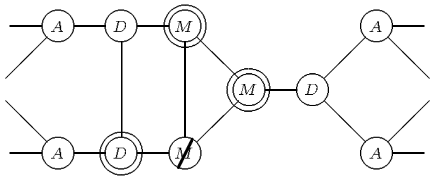
- Half edges connect either to other half edges, or to other vertices in another part of the graph. No connection is permitted that introduces a loop or parallel edges into the graph.

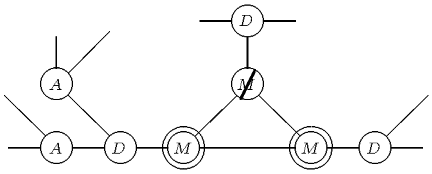
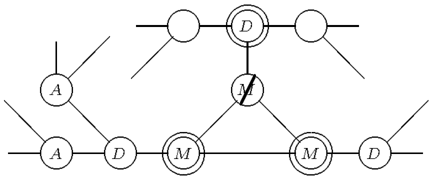
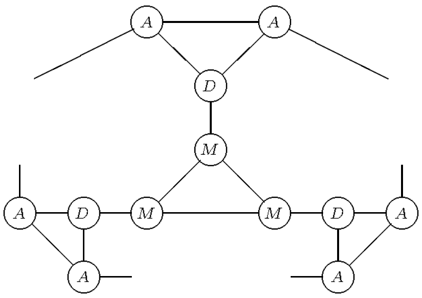
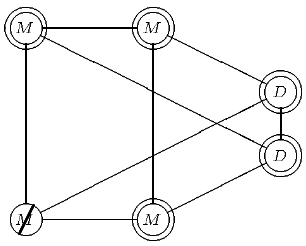
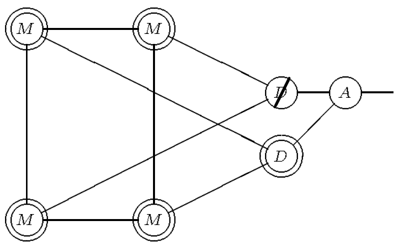
- An edge shown as a dotted line is optional. Its presence or absence doesn't affect the validity of the reduction. Since each vertex has degree at most 3, it is usually the case that not all the dotted lines in a graph can simultaneously represent real edges.

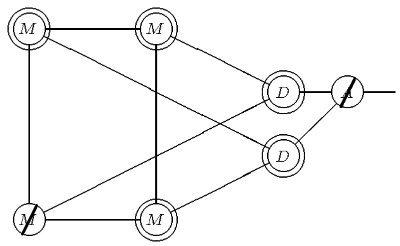
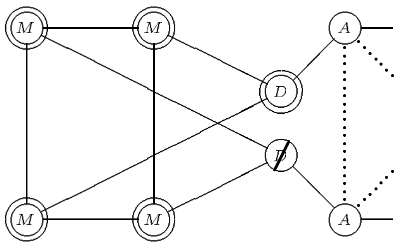
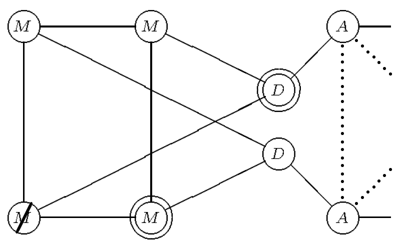
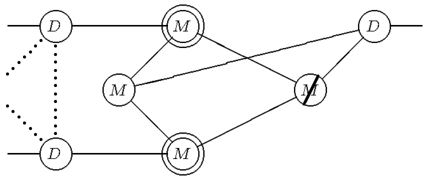
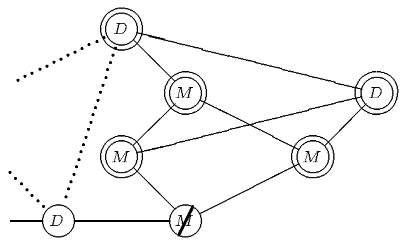
- A vertex with a thick diagonal slash through it is deleted when the reduction is applied. A vertex with a ring around it is used up in some fashion by the reduction. The letters M, D and A are placed in vertices to show which cycle the reduction is based around.

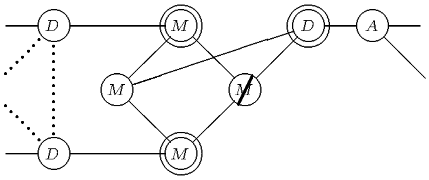
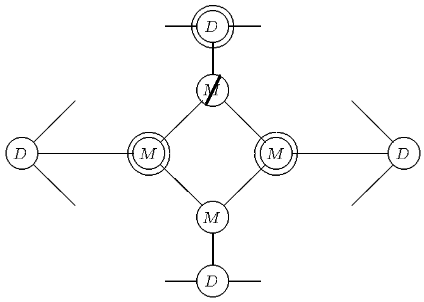
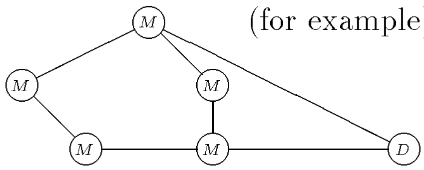
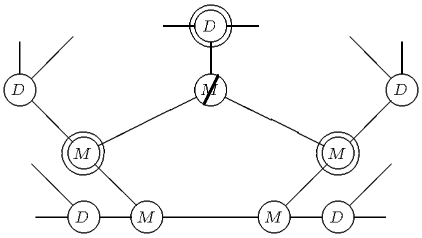
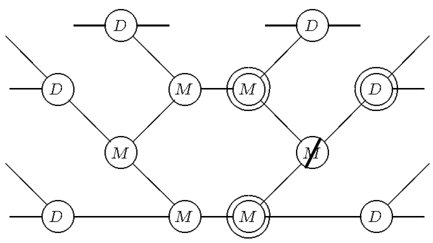
Graph description	**Reduction**	**Ratio**	**Method**
Reduction 1: $M = 3,\ D = 0$		0	base ○
Reduction 2: $M = 3,\ D = 1$		$\frac{1}{4}$	base ○
Reduction 3: $M = 3,\ D = 2$		$\frac{1}{4}$	disjoint ○
Reduction 4: $M = 3,\ D = 2$, D-vertices non-adjacent		$\frac{1}{3}$	bridge
Reduction 5: $M = 3,\ D = 2$, D-vertices adjacent case (i)		$\frac{1}{3}$	base ○
Reduction 6: $M = 3,\ D = 2$, D-vertices adjacent case (ii)		$\frac{2}{7}$	base ⦶

Graph description	**Reduction**	**Ratio**	**Method**
Reduction 7: $M = 3,\ D = 3,\ A = 0$		$\frac{1}{6}$	base ⦶
Reduction 8: $M = 3,\ D = 3,\ A = 1$ Exactly two edges between D-vertices		$\frac{1}{7}$	base $\ominus$
Reduction 9: $M = 3,\ D = 3,\ A = 1$ Exactly two edges between D-vertices		$\frac{2}{7}$	base $\bigcirc$
Reduction 10: $M = 3,\ D = 3,\ A = 2$ Exactly two edges between D-vertices		$\frac{1}{4}$	base $\ominus$
Reduction 11: $M = 3,\ D = 3,\ A = 2$ Exactly two edges between D-vertices		$\frac{1}{3}$	bridge
Reduction 12: $M = 3,\ D = 3,\ A = 1$ One edge between D-vertices		0	degree-2 vertex

Graph description	**Reduction**	**Ratio**	**Method**
Reduction 13: $M = 3,\ D = 3,\ A = 2$ One edge between D-vertices		$\frac{1}{4}$	base ⦶
Reduction 14: $M = 3,\ D = 3,\ A = 3$ One edge between D-vertices case (i)		$\frac{1}{6}$	disjoint ⦶
Reduction 15: $M = 3,\ D = 3,\ A = 3$ One edge between D-vertices case (i)		$\frac{1}{3}$	bridge
Reduction 16: $M = 3,\ D = 3,\ A = 3$ One edge between D-vertices case (ii)		$\frac{1}{4}$	2 bridges
Reduction 17: $M = 3,\ D = 3,\ A = 4$ One edge between D-vertices		$\frac{1}{4}$	2 bridges

Graph description	**Reduction**	**Ratio**	**Method**
Reduction 18: $M = 3,\ D = 3$ No edges between D-vertices		$\frac{1}{3}$	bridge
Reduction 19: $M = 3,\ D = 3$ No edges between D-vertices and some D-vertex not in a $\triangle$		$\frac{1}{4}$	2 bridges
Reduction 20: $M = 3,\ D = 3$ No edges between D-vertices and every D vertex in a $\triangle$		—	no simple reduction with ratio $\leq 1/4$
Reduction 21: $M = 4,\ D = 2,\ A = 0$ In this reduction, and in all following cases, we assume that the graph is $\triangle$-free.		$\frac{1}{6}$	base ⦶
Reduction 22: $M = 4,\ D = 2,\ A = 1$		$\frac{1}{6}$	disjoint ⦶

Graph description	**Reduction**	**Ratio**	**Method**
Reduction 23: $M = 4,\ D = 2,\ A = 1$		$\frac{2}{7}$	base ○
Reduction 24: $M = 4,\ D = 2,\ A = 2$		$\frac{1}{6}$	disjoint ⦶
Reduction 25: $M = 4,\ D = 2,\ A = 2$		$\frac{1}{3}$	bridge
Reduction 26: $M = 4,\ D = 3$		$\frac{1}{3}$	2 bridges
Reduction 27: $M = 4,\ D = 3$ case (i)		$\frac{1}{6}$	disjoint ⦶

Graph description	**Reduction**	**Ratio**	**Method**
Reduction 28: $M = 4,\ D = 3$ case (ii)		$\frac{1}{4}$	3 bridges
Reduction 29: $M = 4,\ D = 4$		$\frac{1}{4}$	3 bridges
Reduction 30: $M = 5,\ D < 5$ Must be covered by an earlier case	 (for example)	—	has a 4-cycle
Reduction 31: $M = 5,\ D = 5$		$\frac{1}{4}$	3 bridges
Reduction 32: $M > 5$ Apply reductions similar to those used in the case $M = 5$		$\frac{1}{4}$	3 bridges